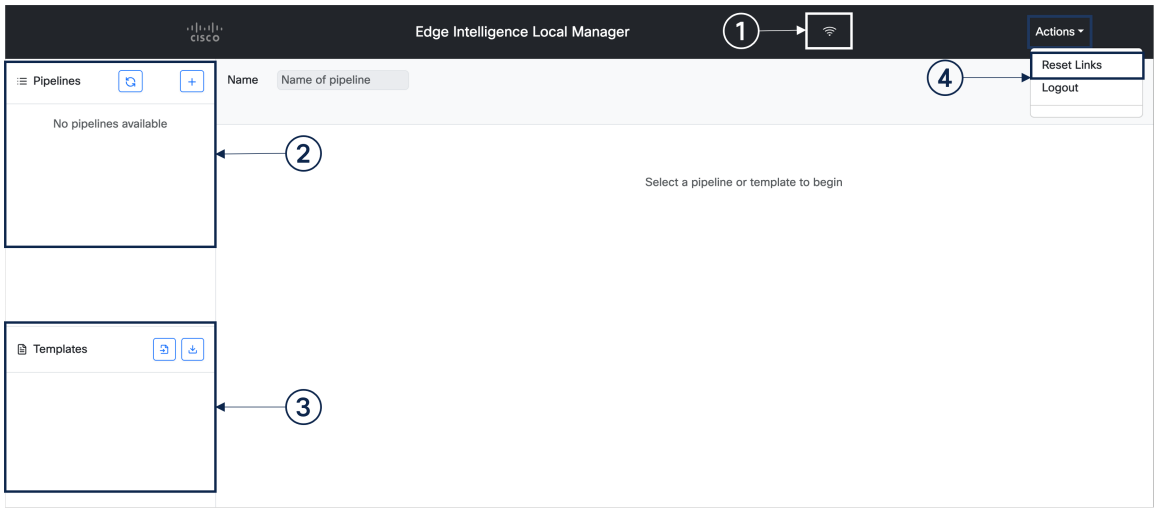


Cisco Edge Intelligence Local Manager

In Cisco Edge Intelligence Local Manager dashboard, we create pipelines to define the data progression.

Along with pipeline creation, the following features are available in the dashboard.

Figure 1: Cisco Edge Intelligence Local Manager dashboard



1. Check Cisco Edge Intelligence connection status:

   Hover over the network connection icon (Wi-Fi icon) in the top banner to view whether Cisco Edge Intelligence is online or offline, and for agent details such as version and ID.

2. Create and view deployed pipelines:

   - The pipelines area provides a quick view of the deployed pipelines and their respective statuses.
   - Click an existing pipeline to view its health status details, or to edit the pipeline's configurations.
   - To create a new pipeline, click the plus (+) icon.

3. View, import, or export templates:

   The templates area displays all the templates that are available in the Cisco Edge Intelligence Local Manager.

   - Click the import icon to upload template files from your local system.

- Click the download icon to download one or all the templates to your local system, in JSON format. You can then import the templates into other agents for deployment.

4. Delete all pipelines:

   To delete all the pipelines in your Cisco Edge Intelligence Local Manager, choose **Actions** > **Reset Links** from the top banner.

Caution Once deleted, a pipeline cannot be retrieved.

Templates are browser-specific and access is restricted to your user credentials. However, deployed pipelines do not have similar restrictions. Multiple users can check the health status of a pipeline and edit any existing pipeline configurations.

- How to create pipelines, on page 2
- Data sources, on page 3
- Add data destinations, on page 24
- About Data Policies, on page 35
- Deploy or undeploy pipelines, on page 58
- View health status, on page 59

How to create pipelines

Creating a Cisco Edge Intelligence pipeline involves defining:

1. One/multiple source asset type
2. One data destination
3. A data transformation method, in the form of a data rule or a data logic

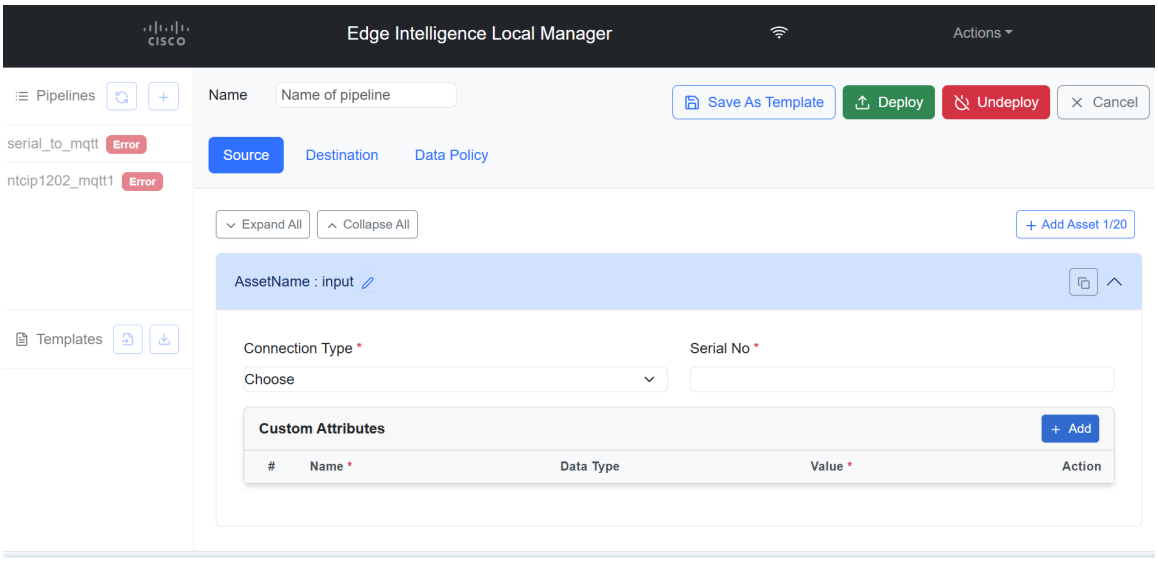
To create a pipeline in the Cisco Edge Intelligence Local Manager GUI, in the **Pipelines** area of the left pane, click the plus (+) sign button. At the top of the page, enter a name for the pipeline. Ensure that each pipeline name for the Cisco Edge Intelligence must be unique, in compliance with the following recommendations.

- Do not use special characters for a pipeline name, as special characters are removed during internal processing. We recommend using the CamelCase naming convention.

  For example, use `WaterSensorSalinityJ2345` instead of `Water Sensor-Salinity_J2345`.

- Avoid ending a pipeline name with the letter `s` to indicate a plural form, as this name can cause internal naming conflicts.

  For example, use `ColdStoragePlc` instead of `ColdStoragePlcs`.



© 2025 Cisco Systems, Inc.

Data sources

Define assets or data source types based on the communication protocols they use. Each protocol then allows further configurations to define the data sources.

In the **Source** tab, these fields are the required for all type of connection types.

Field	Description
Asset Name	Click the pencil icon to enter a name for the asset. This name gives a significant value to the data logic.
Connection Type	From the drop-down list, choose one of the following protocols: • MQTT • Modbus – Serial • Modbus – TCP/IP • OPC-UA • Serial • RSU • NTCIP1202 • NTCIP1203 • NTCIP1204
Serial Number	Enter a serial number for the chosen source type.

Field	Description
Custom Attribute Configuration	You can add custom attributes along with each asset type-specific attribute. Add the following details to configure the custom attribute: **1.** Name: Enter a name for the custom attribute. **2.** Data Type: From the drop-down list, select a data type, string, double, encrypted string, or file. **3.** Value: Enter a custom attribute value. If the Data Type is selected as File: • There can be only one attribute type File for a given asset type. • The max size of the uploaded file should be 12 KB. • The file can be of any type - ASCII or binary. • If the asset is part of a data rule policy and the destination is configured to send the custom attribute to the northbound destination, the base64 equivalent of the file contents will be sent. • If the asset is part of a data logic policy, a custom attribute value is available in the data logic script as a byte array (UInt8Array), and it can be converted to the original format for access. • For example, if the custom attribute "reference_data" was of File type and the uploaded file was as ASCII file, the following code shows how to convert the value available as UInt8Array into ASCII text. `function on_update() {` `...` `var file_contents = new TextDecoder().decode(input.reference_data);` `// Converts UInt8Array to ASCII` `...` `}` Click **Add** to add more custom attributes to any asset configuration

Note Cisco Edge Intelligence Local Manager have the provision to configure the multiple assets under a single pipeline. You can configure upto 20-assets type details and run the deployment.

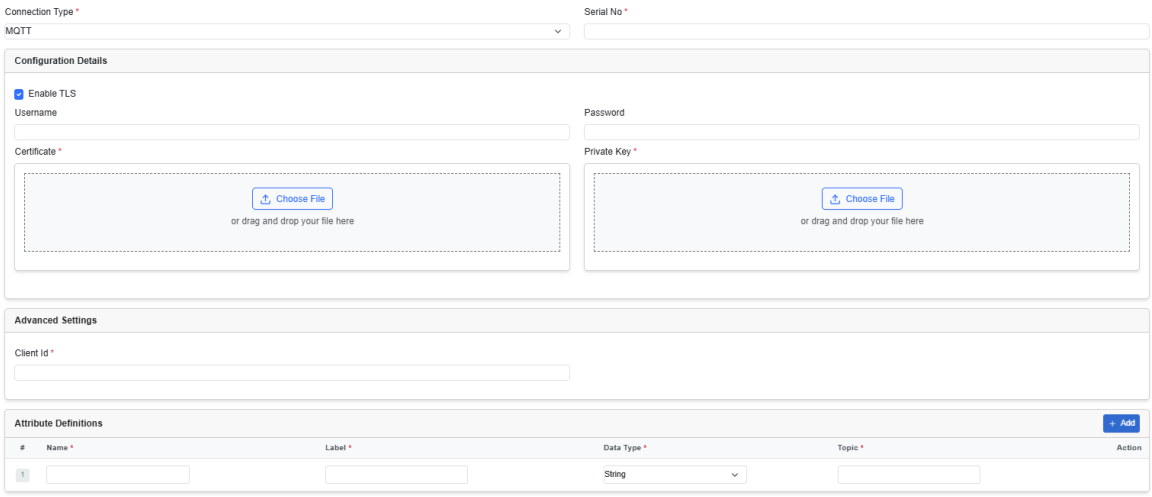
If you want to duplicate an existing asset details with a different serial number, click copy-paste icon next to the asset name.

Configure MQTT asset type

The MQTT asset type enables and configures the Cisco Edge Intelligence MQTT Server. You can publish data to the EI Agent from an MQTT client on the following ports after deploying this to an EI Agent:

- Port 8883 for TLS configurations
- Port 1883 for non-TLS configurations

In the **Source** tab, from the **Connection Type** drop-down list, choose **MQTT**. Then, fill out the following fields.



Field	Description
Configuration Details	
Enable TLS	Choose this checkbox to enable TLS. When you choose to enable TLS, additional fields are displayed to allow the upload of certificate and private key files. The MQTT server that is deployed on the EI agent, uses these certificates and private keys to authenticate the MQTT clients connecting to it.
Username	Enter a username. The connecting MQTT Clients (sensors) will use this username for authentication at the MQTT Server which will be spawned on the EI Agent.

Field	Description
Password	Enter the Password. The connecting MQTT Clients (sensors) will use this password for authentication at the MQTT Server which will be spawned on the EI Agent.
Advanced Settings	
Client ID	Add a client ID to publish data. (Topic-based device or sensor identity detection is not supported). The client ID, specified in the MQTT client connection, differentiates various MQTT connections to Cisco Edge Intelligence.
Attribute Definitions	The data model explains how data is represented in the asset, and what MQTT topics the asset should be updated from. To define a data model, add the following details: 1. **Name**: Enter a name for the data model 2. **Label**: Enter a label for the data model 3. **Data Type**: From the drop-down menu, select a measurement entity (string, integer, float, long, or boolean) for the MQTT topic. 4. **Topic**: Enter the MQTT topic over which the measurement is sent by the transmitting MQTT client. For example, `sensors/tempXY/temp`. Click **Add** to add more data model attributes to the configuration



Note The MQTT topic used to publish sensor data and the data format should match the **data model** JSON file.

MQTT data model example:

```
{
    "apiVersion": 1.0,
    "connectionType": "MQTT",
    "fields": {
      "temperature": {
        "category": "TELEMETRY",
        "label": "Temperature",
        "description": "Outside temperature sensor XY | Temperature",
        "datatype": "Float",
        "topic": "sensors/tempXY/temp"
       },
       "humidity": {
         "category": "TELEMETRY",
         "label": "Humidity",
         "description": "Outside temperature sensor XY | Humidity",
```

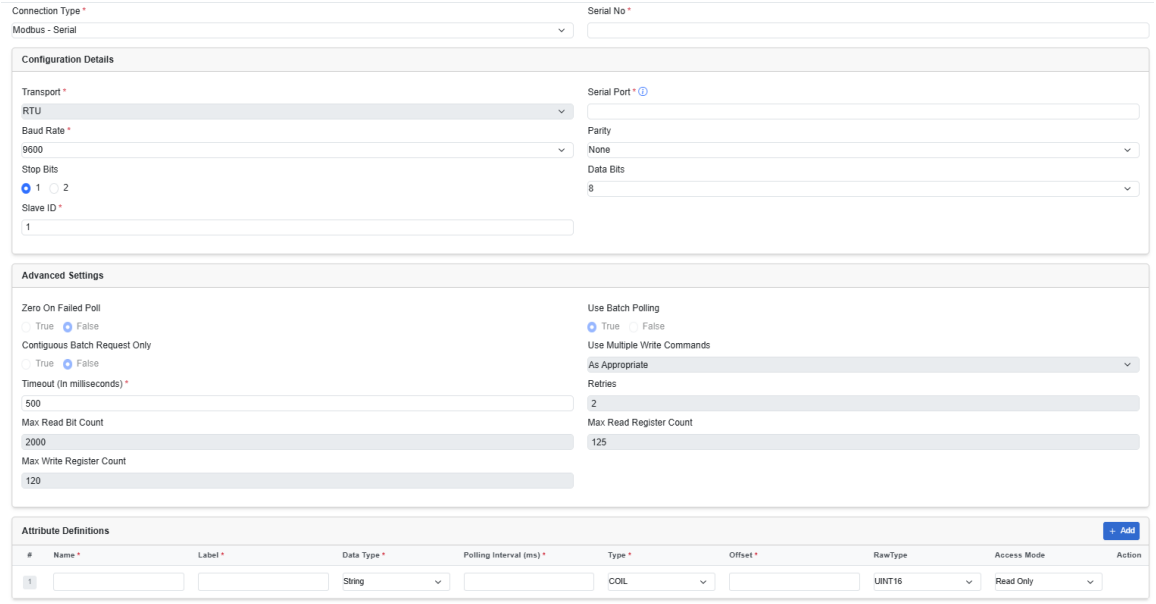
```
            "datatype": "Float",
            "topic": "sensors/tempXY/hum"
          },
          "attr1": {
            "category": "ATTRIBUTE",
            "label": "Attribute 1",
            "datatype": "Float",
            "description": "My Attribute 1",
            "required": true,
            "defaultValue": 12.9
          },
          "attr2": {
            "category": "ATTRIBUTE",
            "label": "Attribute 2",
            "datatype": "String",
            "description": "My Attribute 2",
            "required": false,
            "defaultValue": null
          },
          "encrypted_attr3": {
            "category": "ATTRIBUTE",
            "label": "New Attribute",
            "datatype": "EncryptedString",
            "description": "My Attribute 3",
            "required": true,
            "defaultValue": null
          }
      }
}
```

MQTT topic and sensor data used by MQTT Client example:

```
MQTT Client Topic: sensors/tempXY/hum
MQTT Client Data: 50.0
```

Configure Modbus-Serial asset type

From the **Device Type** drop-down list, choose **MODBUS-Serial**. Then enter the required details in the following fields.



Field	Description
Configuration Details	
Transport	This field is not editable. It has a default value set for RTU.
Serial Port	Enter a port number. For example, `/dev/ttyS0`.
Baud Rate	Enter a baud rate.
Parity	From the drop-down list, choose **None**, **Odd**, or **Even**.
Stop Bits	This value can be either 1 or 2. Choose the required radio button.
Data Bits	From the drop-down list, choose a value 5–9.
Slave ID	This refers to the unique identifier assigned to a Modbus secondary device (such as a sensor or PLC) on a serial network. Modbus protocol requires each secondary device to have a unique ID (1–247) to distinguish it on a shared bus. This ID ensures that the Modbus primary (such as the Cisco EI agent) sends requests to the correct device and processes its responses.
Advanced Settings	
Zero or Failed Poll	The value if there is no response from the asset. This field is not editable.
Use Batch Polling	To request batch responses from the asset. This field is not editable.

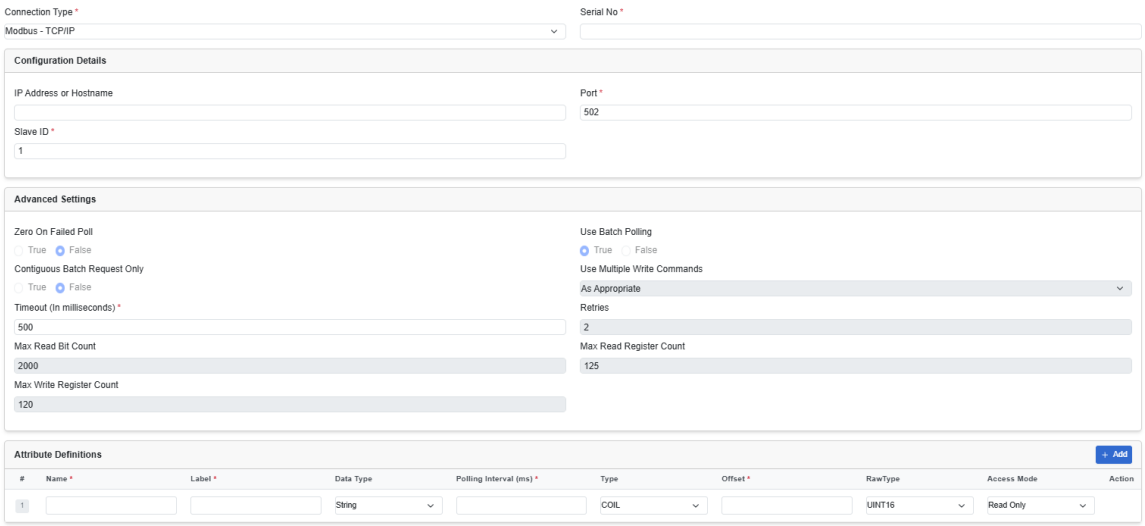
Field	Description
Contiguous Batch Request Only	To request contiguous batch responses from the asset. This field is not editable.
Use Multiple Write Commands	This field is not editable.
Timeout	The time in which to receive the data before it is reset. The default value is 500.
Retries	The number of times the server requests for retransmission of data. This field is not editable.
Max Read Bit Count	The maximum number of bits that the server reads in one read request. This field is not editable.
Max Read Register Count	The maximum number of registers that the server reads in one read request. This field is not editable.
Max Write Register Count	The maximum number of registers that the server writes in one write request. This field is not editable.
Attribute Definitions	A data model explains how data is represented in the asset. You can create a data model using a JSON file. Select one of the following: 1. **Name**: Enter a name for the data model. 2. **Label**: Enter a label value. 3. **Data Type**: From the drop-down list, select a measurement entity (string, integer, float, long, or boolean) for the MODBUS serial data. 4. **Polling Interval (ms)**: Enter a value to define the fastest rate at which the server must test and debug. 5. **Type**: From the drop-down list, choose a data type for industrial control of factory devices. The available choices are **COIL**, **HOLDING**, **DISCRETE**, and **INPUT**. 6. **Offset**: Enter a value to reference a specific register within the function. 7. **RawType**: From the drop-down list, choose the value type for storing binary data or byte strings. 8. **Access Mode**: From the drop-down list, choose an access mode. The available choices are **Read Only**, **Read & Write**, and **Write Only**. Click **Add** to add more data model attributes to the configuration

Modbus_serial data model example:

```
{
  "apiVersion": 1,
  "connectionType": "MODBUS_SERIAL",
  "fields": {
    "pressureInPascal": {
      "label": "MetrLabeModb1",
      "pollingInterval": 50,
      "offset": 12,
      "type": "HOLDING",
      "datatype": "Float",
      "rawType": "VARCHARSTRING",
      "description": "",
      "access": "Write"
    },
    "TemperatureInDegrees": {
      "label": "Temperature",
      "pollingInterval": 41,
      "offset": 56,
      "type": "DISCRETE",
      "datatype": "String",
      "rawType": "INT16",
      "description": "",
      "access": "ReadWrite"
    },
    "HumidityInDegrees": {
      "label": "Humidity",
      "pollingInterval": 20,
      "offset": 45,
      "type": "COIL",
      "datatype": "Int",
      "rawType": "FLOAT64",
      "description": "Humidity Value ",
      "access": "Read"
    }
  }
}
```


Adding Asset Types for Modbus TCP Connection Type

After you select the MODBUS-TCP/IP, complete the following additional fields by adding the details on it.



Field	Description
Configuration Details	
IP Address or Host Name	Enter an IP address or hostname.
Port	Enter a port number.
Slave ID	This refers to the unique identifier assigned to a Modbus secondary device (such as a sensor or PLC) on a serial network. Modbus protocol requires each secondary device to have a unique ID (1–247) to distinguish it on a shared bus. This ID ensures that the Modbus primary (such as the Cisco EI agent) sends requests to the correct device and processes its responses.
Advanced Settings	
Zero on Failed Poll	The value if there is no response from the asset. This field is non-editable.
Use Batch Polling	To request batch responses from the asset. This field is non-editable.
Contiguous Batch Request Only	To request contiguous batch responses from the asset. This field is non-editable.
Use Multiple Write Commands	This field is non-editable.
Timeout	The time set to receive the data before it is reset. The default value is 500.
Retries	The number of times the server requests for retransmission of data. This field is non-editable.
Max Read Bit Count	The maximum number of bits that the server reads in one read request. This field is non-editable.
Max Read Register Count	The maximum number of registers that the server reads in one read request. This field is non-editable.
Max Write Register Count	The maximum number of registers that the server writes in one write request. This field is non-editable.

Field	Description
Attribute Definitions	A data model explains how data is represented in the asset. You can create a data model using a JSON file. Enter the following details: 1. **Name**: Enter a name for the data model. 2. **Label**: Enter a label value. 3. **Data Type**: From the drop-down menu, select a measurement entity (string, integer, float, long, or boolean) for the MODBUS-TCP/IP data. 4. **Polling Interval**: Enter a value to define the fastest rate at which the server should test and debug. 5. **Type**: From the drop-down menu, choose a data type for industrial control of factory devices. The available choices are **COIL**, **HOLDING**, **DISCRETE**, and **INPUT**. 6. **Offset**: Enter a value to reference a specific register within the function. 7. **RawType**: From the drop-down menu, choose the value type for storing binary data or byte strings. 8. **Access Mode**: From the drop-down list, choose an access mode. The available choices are **Read Only**, **Read & Write**, and **Write Only**. Click **Add** to add more data model attributes to the configuration

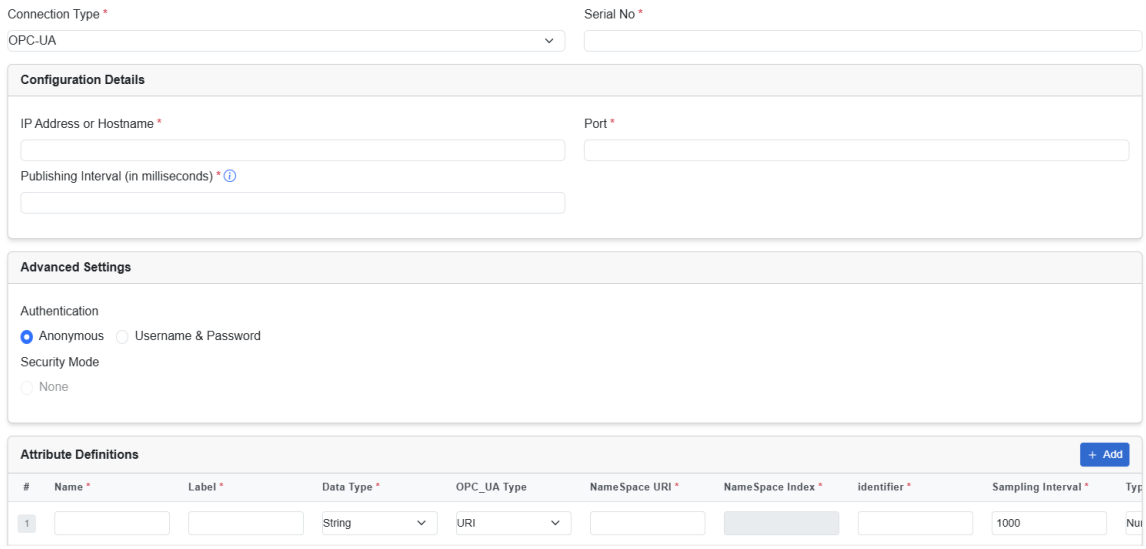
The following is an example of a data model for Modbus-TCP/IP source type.

```
{
  "apiVersion": 1,
  "connectionType": "MODBUS_TCP",
  "fields": {
    "desired_temp": {
      "label": "Desired Temperature",
      "datatype": "Int",
      "description": "WO",
      "rawType": "UINT16",
      "type": "HOLDING",
      "pollingInterval": 5000,
      "offset": 5,
      "category": "TELEMETRY",
      "access": "Write"
    },
    "temp_to_display": {
      "label": "Temperature to be displayed",
      "datatype": "Int",
      "description": "RW",
      "rawType": "UINT16",
```

```
      "type": "HOLDING",
      "pollingInterval": 5000,
      "offset": 100,
      "category": "TELEMETRY",
      "access": "ReadWrite"
    },
    "temp": {
      "label": "Current Temperature",
      "datatype": "Int",
      "description": "RO",
      "rawType": "UINT16",
      "type": "HOLDING",
      "pollingInterval": 5000,
      "offset": 1,
      "category": "TELEMETRY",
      "access": "Read"
    }
  }
}
```

Configure OPC-UA asset type

After you select the asset type OPC-UA, enter the required details in the following fields:



Field	Description
Configuration Details	
IP Address or Host Name	Enter an IP address or hostname.
Port	Enter a port number.

Field	Description
Publishing Interval	This is the requested publishing frequency from the OPC-UA Server. The interval must be greater than or equal to 1000ms. **Note** The OPC-UA server publishing frequency is independent of the metric-specific sample interval. In case the sampling interval (in the following attribute table) of an individual metric is smaller than the publishing interval, the OPC-UA server queues up and send all the sampled values for a metric between the last publish and the current publish.
Advanced Settings	
Authentication	Choose an authentication type: • **Anonymous**: The OPC-UA client inside the EI Agent does not authenticate at the OPC-UA server. Use this authentication type if your OPC-UA server does not have authentication that is enabled for connecting clients. • **Username & Password**: Enter the username and password that the EI Agent must use to authenticate at the OPC-UA server.
Security Mode	This field is non-editable. The default value is **None**.

Field	Description
Attribute Definitions	A data model explains how data is represented in the asset. 1. **Name**: Enter a name for the data model. 2. **Label**: Enter a label for the data model. 3. **Data Type**: From the drop-down menu, select a measurement entity (string, integer, float, long, or boolean) for the OPC-UA data. 4. **OPC_UA Type**: From the drop-down menu, choose **URI** or **INDEX**. 5. **NameSpace URI**: Enter a value to identify the naming authority that defines the identifiers of Node IDs. **Note** This field is editable if you select **URI** under **OPC_UA Type**. 6. **NameSpace Index**: Enter a value to identify the naming authority that defines the identifiers of Node IDs. **Note** This field is editable if you select **INDEX** under **OPC_UA Type**. 7. **Identifier**: Enter a value that is unique across different naming authorities. 8. **Sampling Interval**: Enter a value to indicate the fastest rate at which the server should sample its underlying source for data changes. 9. **Type**: From the drop-down menu, choose the data type for the identifier. The available choices are **Numeric** and **String**. Click **Add** to add more data model attributes to the configuration

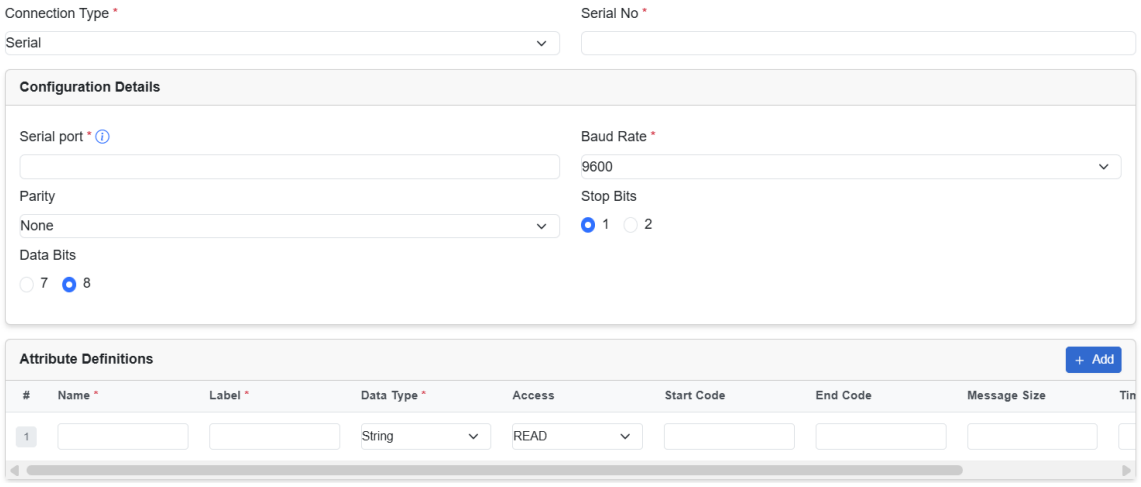
The following is an example of a data model for OPC-UA source type.

```
{
  "apiVersion": 1,
  "connectionType": "OPC_UA",
  "fields": {
    "temperature": {
      "label": "Temperature",
      "description": "",
      "datatype": "Float",
      "nodeId": {
```

```
        "namespaceUri": "2",
        "identifier": "2",
        "type": "numeric"
      },
      "samplingInterval": 1000,
      "category": "TELEMETRY"
    }
  }
}
```

Configure Serial asset type

After you select the asset type **Serial**, enter the required details in the following fields.



Field	Description
Configuration Details	
Serial Port	Enter a port number. For example, `/dev/ttyS0`
Baud Rate	From the drop-down menu, choose 9600, 19200 or you can add a new custom value on it.
Parity	From the drop-down menu, choose **None**, **Odd**, or **Even**
Stop Bits	Choose the radio button for **1** or **2**.
Data Bits	From the drop-down menu, choose **7** or **8**.

Field	Description
Attribute Definitions	A data model explains how data is represented in an asset. 1. **Name**: Enter a name for the data model. 2. **Label**: Enter a label for the data model. 3. **Data Type**: From the drop-down menu, choose **String** or **Binary**. If you select **Binary**, the data is delivered in binary form 1. a. For a data logic policy, the data is delivered as a binary buffer in the data logic script in the *on_update()* function. b. For a data rule policy, the data is sent to the northbound destination in the base64 format. 4. **Access**: From the drop-down menu, choose read, write, or read and write. 5. **Start Code**: Enter the marker that indicates the start of a stream of bytes. 6. **End Code**: Enter the marker that indicates the end of a stream of bytes. 7. **Message Size**: This is the size, in bytes, between the start code and the end code. 8. **Timeout**: Enter a time, in milliseconds, within which to receive data, before it is reset. This is enabled only for **READ** and **READWRITE** access modes. This field is mandatory only if the message size is configured.

Common errors and troubleshooting

Errors can easily occur when configuring a serial port. For example:

- The wiring must be accurate. For example, see the Cisco IR829 installation guide.
- The serial relay service should be configured correctly for the Guest OS. For example, see the Cisco Catalyst IR1101 documentation and Cisco Catalyst IR1800 documentation.
- The physical serial port must be correctly exposed to IOx through the Local Manager.

To troubleshoot a serial interface:

- Make sure that serial port is configured in propagate mode at the IOS level. A current workaround is to use just the 0x prefix as the StartCode to specify an empty StartCode.
- For testing interface options only:
  - Use a data model with a fixed message size of 1 byte and no start code.

- Verify that there is some data that is coming in to ensure that the connection is working.
- Once this is done, the actual data model can be defined.

Serial Connector data model

A serial connector asset type has a reduced data type.

The following combinations are allowed:

- One read attribute
- One read plus one write attribute
- One read-write attribute

Read attribute allows the following configuration combinations:

- Message Size and Timeout
- StartCode and Message Size and an optional Timeout
- StartCode and EndCode and an optional Timeout

Start-/End-Code prefix handling:

- Prefix 0x allows to specify hex encoded binary data. For example, 0x1310 -> CR+LF)
- **Prefix** allows to specify **as-is**

Serial data model example:

```
{
  "apiVersion": 1,
  "connectionType": "SERIAL",
  "fields": {
    "data_string1": {
      "label": "My Data String",
      "datatype": "String",
      "description":    "serial read attribute",
      "access": "READ"
      "startCode": "$",
      "endCode": "0x0a",
      "messageSize":""
    }
  }
}
```

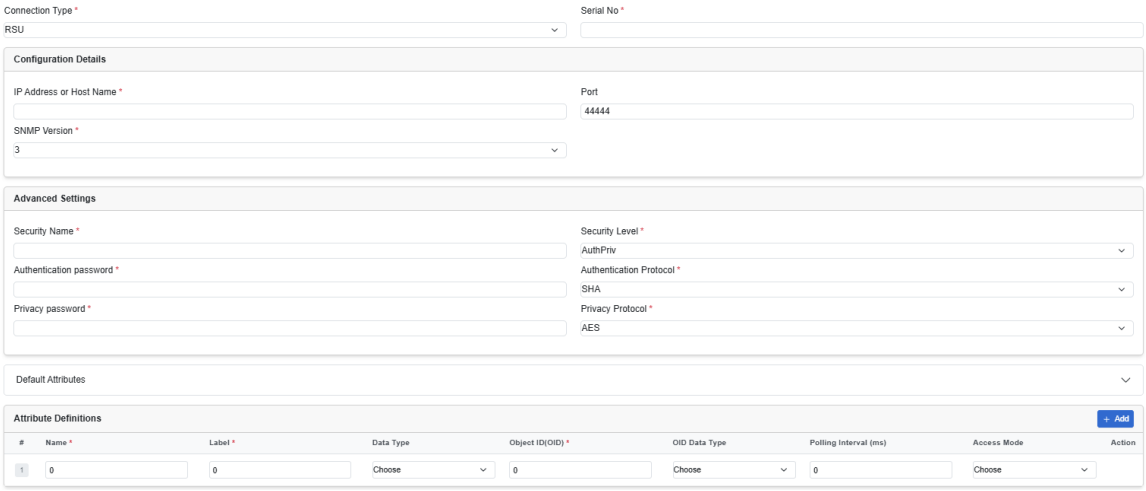
RSU asset type settings

The RSU asset type supports a set of static attributes in addition to the regular configurable attributes. Static attributes are always available and not required to be configured.

- The static attributes have a JSON string content and reflect incoming DSRC messages, except storeAndRepeatMessage and broadcastImmediately static attributes.
- The storeAndRepeatMessage must be set as an array of message objects. All previous messages will be overwritten by the new array.

- The current array of messages can be obtained by reading the attribute.
- The basic configuration for this asset type includes the host, port, and SNMP version.
- Advanced settings like community or authentication data must be set depending on the SNMP version.

In the **Source** tab, from the **Device Type** drop-down list, choose **RSU**. Then, fill the following fields.



Field	Description
Configuration Details	
IP Address or Host Name	Enter the IP address or hostname.
Port	Enter the port number.
SNMP Version	Select a version from the drop-down list from 1, 2c and 3. 3 is the most secure version.
Advanced Settings	Complete these fields based on the selected SNMP Version.
Default Attributes	It shows the default attributes that are specific to RSU. Even if you will not add any additional attributes, it runs for the agents.

Field	Description
Attribute Definitions	A data model explains how data is represented in an asset. 1. **Name**: Enter a name for the data model. 2. **Label**: Enter a label for the data model. 3. **Data Type**: From the drop-down list, choose **String**, **Integer**, **Float**, **Long**, or **Boolean**. 4. **Object ID (OID)**: OID address is used to uniquely identify managed devices and their statuses. 5. **OID Data Type**: OID is the data type for the object. **String** or **Integer**. 6. **Polling Interval (ms)**: This indicates the fastest rate at which the Server should test and debug. Select a unit from the up-down menu. 7. **Access Mode**: From the drop-down list, choose read, write, or read and write. Click **Add** to add more attributes to the asset configuration.

NTCIP1202, NTCIP1203, NTCIP1204 asset type settings

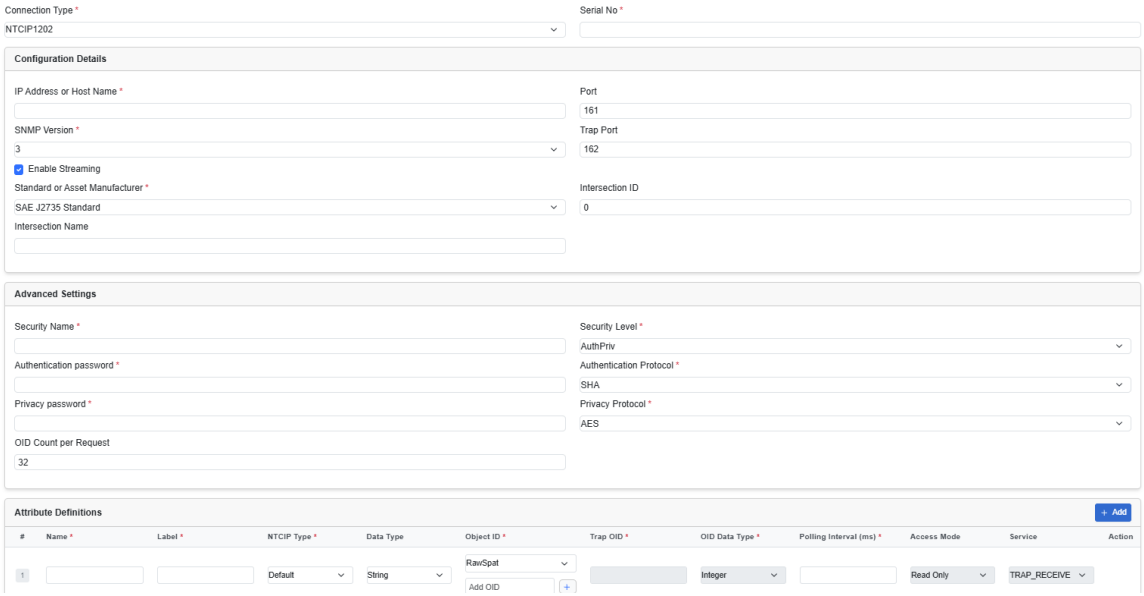
Cisco Edge Intelligence Local Manager supports three NTCIP devices. Use the Asset Type for the correct connection type.

- NTCIP 1202—Actuate Signal Controller
- NTCIP 1203—Dynamic Message Sign
- NTCIP 1204—Road Weather Information System

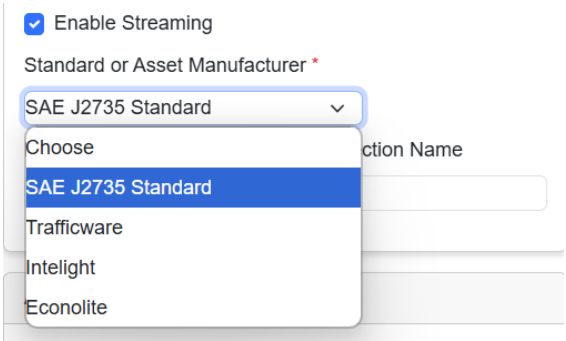
The basic configuration for all these three asset types include the host, port, and SNMP version.

Advanced settings like community or authentication data must be set depending on the SNMP version.

Each NTCIP asset type supports a set of static attributes in addition to the regular configurable attributes.



Field	Description
Configuration Details	
IP Address or Host Name	Provide the IP address or Host Name.
SNMP Version	Select a version from the drop-down list from 1, 2c and 3. 3 is the most secure version.
Port	Provide a NTCIP connector port number. **Note** NTCIP1202 port number should never match Trap port number, and vice versa.
Trap Port (Applicable only for device type: NTCIP1202)	Provide Trap port number. **Note** Trap port number should never match NTCIP1202 port number, and vice versa.

Field	Description
Enable Streaming (Specific to device type: NTCIP1202)	Choose this checkbox to enable Streaming. When you choose to enable Streaming, an additional field **Standard or Asset Manufacturer** protocol is displayed. NTCIP 1202 allows the streaming of data using standard and Asset manufactured protocols. The different protocols are: Standard protocol: • SAE J2735 Standard Asset manufactured protocols • Trafficware • Intelight • Econolite 
Intersection ID (Specific to device type: NTCIP1202)	Intersection ID is optional and depends on the user's specific use case. You can choose to configure this field if required for their particular scenario. For example, in traffic use cases, this information is mandatory.
Intersection Name (Specific to device type: NTCIP1202)	Intersection Name is optional and depends on the user's specific use case. You can choose to configure this field if required for their particular scenario.
Advanced Settings	The specific Advanced Setting details vary based on the SNMP Version. Provide the appropriate authentication information accordingly.

Field	Description
Attribute Definitions	A data model explains how data is represented in an asset. **1.** **Name**: Enter a name for the data model. **2.** **Label**: Enter a label for the data model. **3.** **NTCIP Type**: From the drop-down list, choose **Default** or **trap**. **4.** **Data Type**: From the drop-down list, choose **String**, **Integer**, **Float**, **Long**, or **Boolean**. **5.** **Object ID**: From the drop-down list, choose **RawSpat**, **NTCIP-1211 SRM**, and **SAE J2735 SRM**. Bottom to that add OID. OID address is used to uniquely identify managed devices and their statuses. **Note** This field is applicable if you select **Default** under **NTCIP Type**. **6.** **Trap OID**: Enter an ID for Trap NTCIP type. **Note** This field is applicable if you select **Trap** under **NTCIP Type**. **7.** **OID Data Type**: OID is the data type for the object. **String** or **Integer**. **8.** **Polling Interval (ms)**: It indicates the fastest rate at which the Server tests and debug. Select a unit from the up-down menu. **9.** **Access Mode**: From the drop-down list, choose read, write, or read and write. **Note** This field is applicable if you select **Default** under **NTCIP Type**. **10.** **Service**: From the drop-down list, choose TRAP_RECEIVE. **Note** This field is applicable if you select **Trap** under **NTCIP Type**.

Add data destinations

In the **Destination** tab, define where configured data policies must send data to.

The supported destinations are:

- MQTT servers
- AWS server
- Azure IoT Hub
- Splunk

From the **Type** drop-down list, choose the destination. Then, configure the connection settings for the chosen destination.

Before you configure a data destination in the pipeline, ensure that you have set up the servers or hubs. You must have the required identifying information ready to complete the destination configuration.

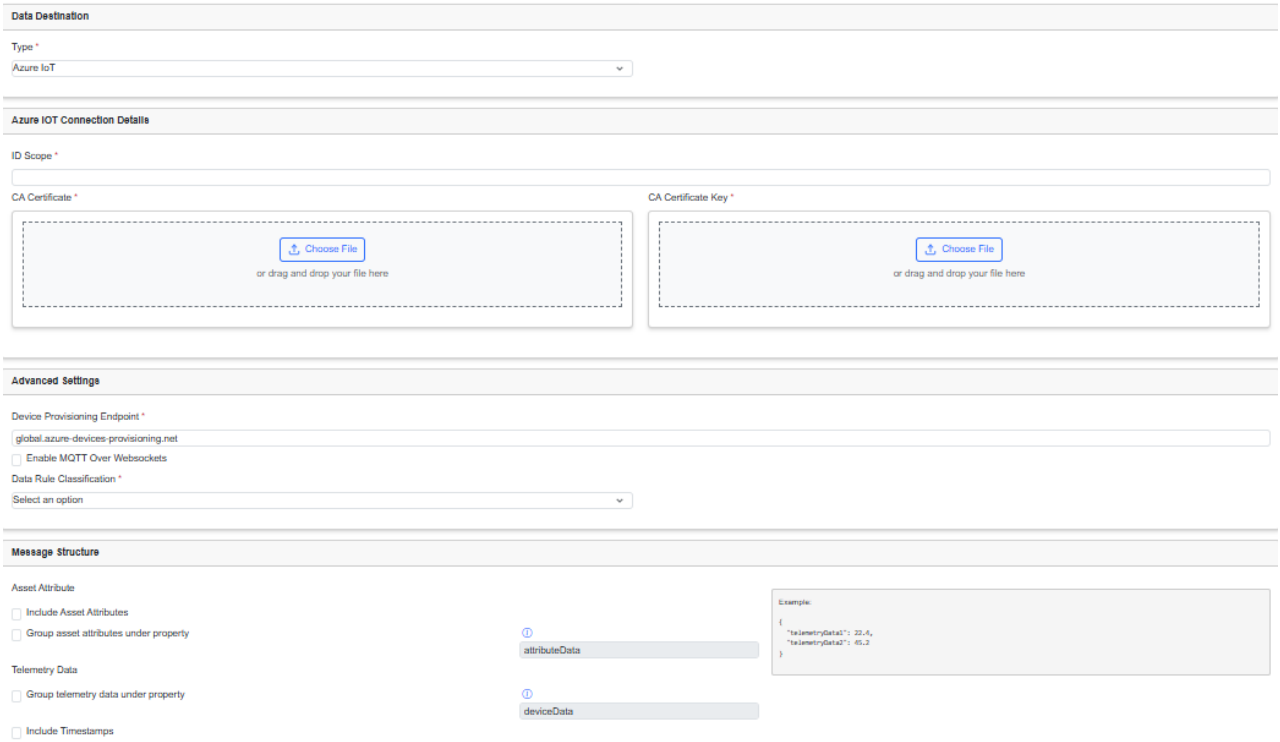
Table 1: Feature History Table

Feature	Release Information	Feature Description
Splunk destination	Release 2.2.x	Splunk is introduced as a new data server destination in this release. Splunk integration serves as a new data server for configuring a pipeline in Cisco Edge Intelligence.

Add a Microsoft Azure IoT Hub destination

Procedure

In the **Destination** tab, from the **Type** drop-down list, choose **AZURE IoT**, and enter the required details in the following fields.



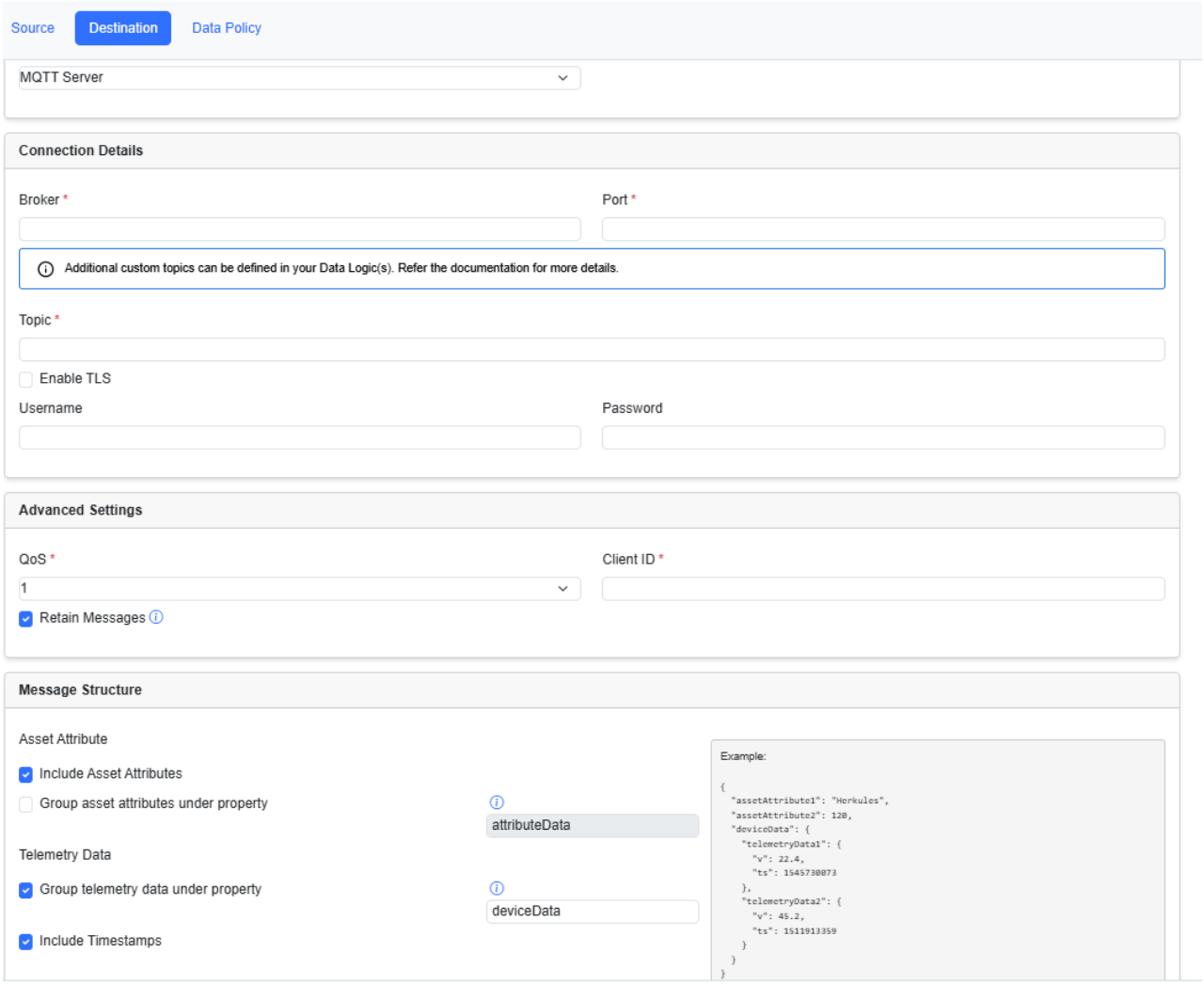
Field	Description
Azure IoT Connection Details	
ID Scope	Enter the ID Scope that is displayed in the **Azure Device Provisioning Service Overview** page.
CA Certificate	Upload the intermediate CA Certificate file that you have configured in your Azure Device Provisioning Service. **Note** The status of this CA certificate in the Azure Provisioning Service must be marked as trusted, or device creation (and therefore metric sending) is not allowed by Azure.
CA Certificate Key	Upload the unencrypted private certificate key file that belongs to the intermediate CA certificate that you uploaded. The private key must be in PKCS8 format, and must not include a passphrase. To convert an existing key, use the command: `openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in azure-iot-test-only.intermediate.key.pem -out azure-iot-test-only.intermediate.pkcs8.pem`
Advanced Settings	
Device Provisioning Endpoint	Enter the **Global Device Endpoint** value from the Azure IoT Hub Device Provisioning Service.

Field	Description
Enable MQTT over WebSockets	Check this check box to enable the browser to leverage all MQTT features.
Data Rule Classification	From the drop-down list, choose **DEVICE_PROPERTY** or **TELEMETRY**.
Message Structure	Choose the required options in this area to customize the structure of the device-to-cloud message. You can choose **Include asset attributes**, **Group asset attributes under property** in the Asset Attributes section or choose **Group telemetry data under property**, **Include Timestamps** in the Telemetry Data section. The data can be sent in a flat structure or can be grouped with a key. Customization does not apply if the chosen data policy is of the type Device Properties.

Add an MQTT Server destination

Procedure

In the **Destination** tab, from the **Type** drop-down list, choose **MQTT Server** and enter the required details in the following fields.



Field	Description
Connection Details	
Broker	Enter the URL or IP address of your MQTT broker.
Port	Enter the port number used by the broker.

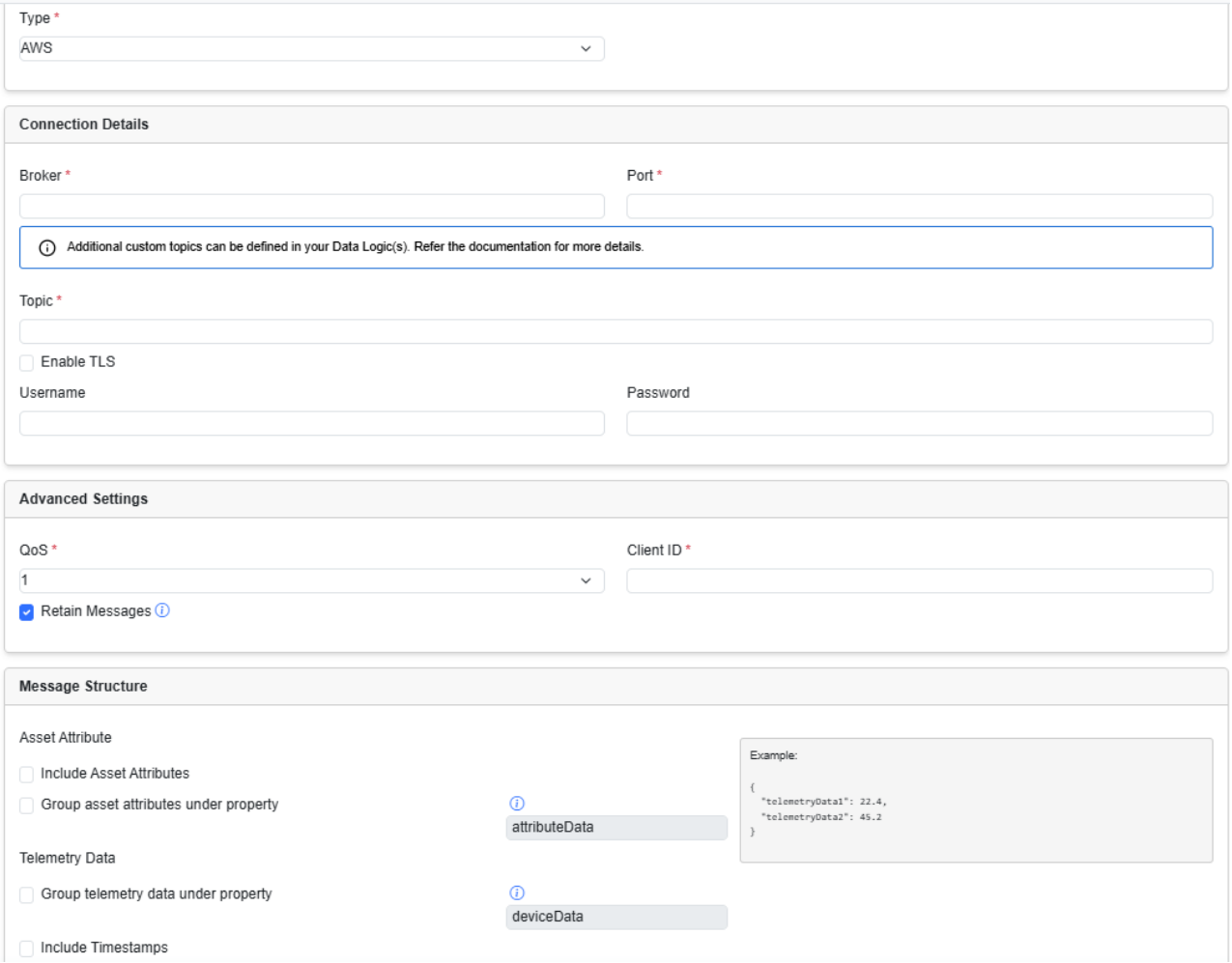
Field	Description
Topic	Enter the topic to which device states and other data are published. For example, cisco/edge-intelligence/telemetry/%deviceSerialNumber%/%deviceSerialNumber%. The example topic matches the device or asset instance serial number that is configured previously in the source tab. **Note** MQTT topic has a restricted number of characters that can be used in a topic name. For example, # or + cannot be part of a topic name. Topic names are URL-encoded to ensure that they do not violate MQTT specifications. URL-encoding also allows northbound applications to decode a topic easily to get to the original contents.
Enable TLS	Check the TLS check box to enable the protocol. When you choose TLS, the following fields are displayed: • **Verify Peer**: Check this checkbox to allow peer verification. When you select this option, the **Certificate** field is displayed where you can upload a CA certificate. • **Enable X.509**: Check this check box to use X.509 certificates. When you select this option, two fields are displayed where you can upload a CA certificate and a private key. Enabling the use of X.509 certificates allows you to turn on configure MQTT brokers (like Mosquitto) to require certificates for authentication. You can also use X.509 client certificates instead of usernames and passwords to ensure that only trusted assets are allowed to send data to a cloud MQTT broker. **Note** • The private key must be PKCS8-compatible. • The generated certificate for each asset contains the CN - Serial Number of the asset. • X.509 certificates can be used with the username and password authentication method, or as the only authentication method.
Username	Enter the username to connect to the MQTT destination broker.
Password	Enter the password to connect to the MQTT destination broker.

Field	Description
Advanced Settings	
QoS	From the drop-down list, choose **0**, **1**, or **2**.
Client ID	Enter a Client ID. The Client ID field is not URL-encoded because there are no restrictions in MQTT specifications about allowed characters.
Retain Messages	Check this check-box to retain messages on the broker for new subscribers.
Message Structure	Choose the required options in this area to customize the structure of the device-to-cloud message. You can choose **Include asset attributes**, **Group asset attributes under property** in Asset Attributes section or choose **Group telemetry data under property**, **Include Timestamps** in Telemetry Data section. The data can be sent in a flat structure or grouped with a key. Customization does not apply if the chosen data policy is of the type Device Properties.
Cloud to Network Device	
Enable cloud to data logic commands	Use the cloud to data logic commands to send a command (with payload) from a cloud app to a data logic on an edge device. The data logic script parses the command. This feature provides bi-directional communication between the cloud and edge, allowing the application to send a command and receive a response. For example, a cold storage unit connected to an edge device can send commands to: • Set the temperature on the cold storage unit • Initiate a defrost action on the cold storage unit To enable cloud to data logic commands, enter the following topic values: **a.** **Command Topic**: Enter the syntax and variables in the format, *cisco/edge-intelligence/**commands**/variable*. **b.** **Response Topic**: The response topic must use the format *cisco/edge-intelligence/**responses**/variable*. Responses are optional. If a script doesn't send a response, nothing is published to the topic.

Add an AWS Server destination

Procedure

From the **Destination** tab, from the **Type** drop-down list, choose **AWS** and enter the required details in the following fields.



Field	Description
Connection Details	
Broker	Enter the URL or IP address of your AWS broker. This info can be found from the AWS IoT **Settings** page
Port	Enter `8883`.
Topic	Enter a topic to which device states and other data are published.

Field	Description
Enable TLS	For AWS destinations, you must enable the TLS protocol. When you choose TLS, the following fields are displayed: • **Verify Peer**: Do not choose this option. • **Enable X.509**: Check this check box to use X.509 certificates. When you select this option, two fields are displayed where you can upload a CA certificate and a private key.
Username	Do not enter any value in this field.
Password	Do not enter any value in this field.
Advanced Settings	
QoS	From the drop-down list, choose **1**.
Client ID	Enter a Client ID.
Retain Messages	For AWS destination type, do not choose the **Retain Messages** option. This option retains messages on the broker for new subscribers.
Message Structure	Choose the required options in this area to customize the structure of the device-to-cloud message. You can choose to include asset attributes, include timestamps, group asset attributes, or group telemetry data in the device-to-cloud messages. The data can be sent in a flat structure or can be grouped with a key. Customization does not apply if the chosen data policy is of the type Device Properties.
Cloud to Network Device	

Field	Description
Enable cloud to data logic commands	Use the cloud to data logic commands to send a command (with payload) from a cloud app to a data logic on an edge device. The data logic script parses the command. This feature provides bi-directional communication between the cloud and edge, allowing the application to send a command and receive a response. For example, a cold storage unit connected to an edge device can send commands to: • Set the temperature on the cold storage unit • Initiate a defrost action on the cold storage unit To enable cloud to data logic commands, enter the following topic values: **a.** **Command Topic**: Enter the syntax and variables in the format, *cisco/edge-intelligence/**commands**/variable*. **b.** **Response Topic**: The response topic must use the format *cisco/edge-intelligence/**responses**/variable*. Responses are optional. If a script doesn't send a response, nothing is published to the topic.

Add a Splunk server destination

Splunk is a software platform designed to collect, analyze, and visualize machine-generated data in real time, delivering operational intelligence for IT operations, security, and business analytics. With the capability in Cisco Edge Intelligence to use Splunk as a destination, customers can seamlessly send their data to Splunk.

Note When you choose Splunk as the destination type, set the data policy type to Data Logic. Data Rule policy type is not supported for this destination.

This procedure explains the fields and settings for the Splunk server destination.

Before you begin

Make sure that you have an active Splunk account with the HTTP Event Collector enabled.

Procedure

In the **Destination** tab, select **Splunk** from the **Type** drop-down list. Enter the required details in each field.
Table 1 lists the required configuration fields and settings for the Splunk server destination.

Field	Description
Connection Details	
HEC URL	Enter the complete URL for the HTTP Event Collector (HEC).
HEC token	Enter the HEC token.
For more information about configuring the HEC token and other configuration details, see the HTTP Event Collector Guide.	
Enable TLS	For Splunk destinations, you must enable the TLS protocol. Enabling TLS displays these checkboxes. • Verify Server Certificate: This option enables certificate validation for secure connections. • Enable Mutual TLS (mTLS): This option allows the use of client certificates for authentication.
Verify Server Certificate	Select the checkbox to upload a **CA Certificate Bundle**, which verifies the indexer.
Enable Mutual TLS (mTLS)	Selecting this checkbox displays two fields for uploading a client certificate and client private key. • Client Certificate: Provide a PEM file containing the client certificate. • Client Private key: Provide a PEM file containing the decrypted private key associated with your client certificate.

How to configure single or batch payloads

Splunk data can be sent as a single payload or in batches. This approach is generally used for handling large volumes of data.

This sample snippet demonstrates how to send North Bound data to Splunk.

- Using a single payload

```
function init() {
   logger.info("Starting initialization")

   // SSL setup can be added here if required
}

var counter = 100

function on_update() {
   // Reserved for external update triggers
}

function on_time_trigger() {
```

```
  counter = counter + 1

  // Create a single event payload
  var payload = {
    event: {
      pressure: counter
    },
    host: "FCW22360076",
    source: "FCW22360076",
    sourcetype: "EI Agent"
    // Optional fields like index or timestamp can be added here
  }

  // Send the payload to output immediately
  publish("output", payload);
}
```

- Using a batch payload

```
• function init() {
    logger.info("Starting initialization and setup")
    // SSL options can be added here if needed in future
  }

  // Message buffer to hold event data before sending
  var messageBuffer = []

  // Max number of events to buffer before sending
  const maxBufferSize = 2

  // Example counter for generating event values
  var counter = 100

  function on_update() {
    // Reserved for handling updates from external source
  }

  function on_time_trigger() {
    counter = counter + 1
    // non batch payload
    var payload = {
      "event": {
        "data": counter,
        "escaped_chars": "Line 1\\nLine 2\\tTabbed\\\"Quoted\\\"",
      },
      "host": "FCW22360076",
      "source": "FCW22360076",
      "sourcetype": "EI Agent",
      //"index": "ei-hec-index",
      // "time": new Date(trigger.timestamp).getTime() / 1000
    };
    //publish("output", payload);

    // batch payload
    for (var i = 0; i < 3; ++i) {
      messageBuffer.push(payload);
    }
    publish("output", messageBuffer);
    messageBuffer = [];
  }
```

About Data Policies

Data Policies define how data is sent from edge assets to a destination. There are two types of policies:

- **Data Logic**: Data is transformed before being sent to a destination. Data Logic scripts are developed using Microsoft VS Code and embedded UI editor.
- **Data Rule**: Data is sent from Assets to a destination without transformation.

Restriction Make sure to configure **Data Logic** when you have configured multiple Source type. Proceeding only with **Data Rule**, will pop-up an error while deployment.

Table 2: Feature History Table

Feature	Release Information	Feature Description
HTTP supports in data logic script	Release 2.2.x	HTTP and HTTPS are new enhancement features in the data logic script that allows you to run various methods to fetch data from the source.
Get a device's asset serial number from global device model	Release 2.2.x	Cisco Edge Intelligence data logic scripts now offer enhanced flexibility for accessing a device-specific attribute. You can retrieve asset serial number directly from the global device mode using the `on_update()` function. `function on_update() { serial_number = input.asset_serial_number; serial_number = globalThis[trigger.device_name].asset_serial_number; }`

Scripting engine tutorial

This is a basic user guide on how to implement scripts for Cisco Edge Intelligence. Internally, this feature is based on duktape 2.7.0 and supports the same subset as duktape. It conforms to ES5.0/ES5.1, with semantics updated from ES2015 or later when appropriate.

For additional information, refer to the these links:

- Duktape Programmers Guide
- ECMA Script Language Specification

High level flow for data logic scripts

A data pipeline can be configured to use data logic, such as JavaScript code, to transform data. It is important to understand when the user code is called and what information is accessible. Each data logic instance is called for every change to fields in the input data model. The user code can publish a value for consumption by an egress link or decide not to publish a value.

User code can create global objects that persist across invocations, which allows implementation of stateful transformations such as sliding averages, sliding medians, and histograms. Additionally, a script can be configured for customization and reuse in different data pipelines. For example, the same script can aggregate the last 30 values in one pipeline and 100 values in another. The count of values to aggregate is defined as a configuration parameter.

Required implementations

Each script must provide an `on_update()` function, which is called whenever a field in the input model changes. The function does not return a value. Set the desired outputs on the global `output` object. Outputs are published when `output.publish()` is called.

Built-in global objects

Output

The `output` shall be used to set the desired output values. The names match the output model as specified in the VSCode plugin. For example, if the output model specifies a `foo` field, it has to be set with `output.foo = ...`.

Just setting the output fields is not sufficient. To give the script implementer strict control over when and if to publish the changed output model, he must call `output.publish()` to explicitly publish the transformed value.

Input

The `input` is the representation of the input model. Input model fields are input object fields. On `on_update()` call, `input` contains the input model's last known values. They can simply be accessed with `var x = input.foo`.

Trigger

The `trigger` identifies the field name of the input model that has been changed. Be aware the simultaneous changes to multiple values of the input model are not possible and each change has to be handled on its own. This can be used to access the last changed value: `var x = input[trigger.field_name]`

Parameters

The `parameters` are configurable runtime options for each VSCode pipeline. These settings are to be used to configure the script depending on the pipeline. For example, a script with alerting capabilities has a different threshold per pipeline.

Built-in global functions

Optional initialization

It is optional to provide a `init()` function. If present, this function is called exactly once when the pipeline is incorporated. Instantiation occurs when the pipeline is first created or after each restart.

DSRC encode and DSRC decode

The global functions `dsrc2016_encode` and `dsrc2016_decode` are used to encode and decode messages to and from ASN.1 format.

Encode JSON strings to Uint8Array

```
/**
* @fn dsrc2016_encode(json_string)
* @brief function to encode provided input of json string into Uint8Array format
* @param json_string -> String (message contains json string format)
* @return Uint8Array -> Encoded message in byte array format
*/ dsrc2016_encode(json_string);
```

Decode Uint8Array to JSON string

```
/**
* @fn dsrc2016_decode(buffer)
* @brief function to decode provided input of byte array into json string
* @param buffer -> Encoded message in Uint8Array byte array format
* @return json_string -> String (message contains json string format)
*/
dsrc2016_decode(buffer);
```

BinaryUtils

The `BinaryUtils` is the representation of binary utility methods to operate on binary data. The following methods are part of the `BinaryUtils` object:

Table 3: Binary mask string

Descriptions	Example (Valid)	Example (Invalid)
Must start with the prefix `0b` or `0B`	0b1100	1100
Only the characters `0` and `1` are allowed in the Binary Mask String	0b1010	0b1020
The character _ is allowed as a separator to improve readability and can appear any number of times	0b11_00_11	0b11-00-11
Mask string length (excluding _)depends on API type (For example, `mask16AndShiftRight` expects exactly 16 `0` or `1` digits)	0b_0000_1111_1111_0000	0b_00_1111_1111_0 (length≠16)
The digit `1` must always appear consecutively (all 1's are together, no 0's between them)	0b_0000_1111_0000_0000	0b_0010_0111_1110_0000

Descriptions	Example (Valid)	Example (Invalid)
The mask string cannot consist entirely of `0` characters	0b_0000_0000_0000_0001	0b_0000_0000_0000_0000

Table 4: Hex mask string

Descriptions	Example (Valid)	Example (Invalid)
Must start with the prefix `0x` or `0X`	0xFF0F	FF0F
Allowed characters: `0-9`, `a-f`, `A-F`	0x1A2B	0x1G2B
The character _ is allowed as a separator to improve readability and can appear any number of times	0xFF_FF_00_00	0xFF-FF-00-00
String mask length (excluding _) depends on API type (for example, `mask32AndShiftRight` expects 8 hex characters)	0xFFFF_FFFF	0xFFFF_FF
The mask must represent consecutive 1s in binary (no gaps between 1s in the bit pattern)	0x0000FFF0	0x0000F0F0
The mask string cannot consist entirely of 0 characters	0x00000001	0x00000000

Valid hex string:

- 0x_00_FF_FF_00 => 32 Bit APIs
- 0x_3F_FC => 16 Bit APIs
- 0x_F_8_ => 8 Bit APIs

Invalid hex string:

- 0x_00_F0_FF_00 => 32 Bit APIs because of non-consecutive 1 bit in mask
- 0x_00_FF_FF => 16 Bit APIs because of invalid length
- 0xFG => 8 Bit APIs because invalid character in mask

Functions	Results
mask8 ()	/** * @fn mask8() * @brief gets the number of bit value from the input byte position as according to mask * @param mask -> String (Binary or Hex Mask string. Mask should always have 8 bits/1byte) * @param byte_value -> Uint8 (Input Byte Value) * @return 8 bit integer -> masked bits from byte_value */ / BinaryUtils.mask8(mask, bytes_value);
mask16 ()	/** * @fn mask16() * @brief gets the number of bit value from the input buffer index as according to Mask * @param mask -> String (Binary or Hex Mask string. Mask should always have 16 * bits/2bytes) * @param buffer -> Uint8Array[] (Input buffer) * @param buffer_index -> Integer (Start buffer index) * @return Integer -> masked bits from input buffer */ BinaryUtils.mask16(mask, buffer, buffer_index);
mask32 ()	/** * @fn mask32() * @brief gets the number of bit value from the input buffer index as according to Mask * @param mask -> String (Binary or Hex Mask string. Mask should always have 32 * bits/4bytes) * @param buffer -> Uint8Array[] (Input buffer) * @param buffer_index -> Integer (Start buffer index) * @return Integer -> masked bits from input buffer */ BinaryUtils.mask32(mask, buffer, buffer_index);

Functions	Results
mask8AndShiftRight ()	/** * @fn mask8AndShiftRight() * @brief gets the number of bit value from the input byte position as according to Mask and right * shift the value for all zero bits * @param mask -> String (Binary or Hex Mask string. Mask should always have 8 bits/1byte) * @param byte_value -> Uint8 (Input Byte Value) * @return 8 bit integer -> masked bits from byte_value after right shift */ BinaryUtils.mask8AndShiftRight(mask, bytes_value);
mask16AndShiftRight ()	/** * @fn mask16AndShiftRight() * @brief gets the number of bit value from the input buffer index as according to Mask and right * shift the value for all zero bits * @param mask -> String (Binary or Hex Mask string. Mask should always have 16 * bits/2bytes) * @param buffer -> Uint8Array[] (Input buffer) * @param buffer_index -> Integer (Start buffer index) * @return Integer -> masked bits from input buffer after right shift */ BinaryUtils.mask16AndShiftRight(mask, buffer, buffer_index);
mask32AndShiftRight()	/** * @fn mask32AndShiftRight() * @brief gets the number of bit value from the input buffer index as according to Mask and right * shift the value for all zero bits * @param mask -> String (Binary or Hex Mask string. Mask should always have 32 * bits/4bytes) * @param buffer -> Uint8Array[] (Input buffer) * @param buffer_index -> Integer (Start buffer index) * @return Integer -> masked bits from input buffer after right shift */ BinaryUtils.mask32AndShiftRight(mask, buffer, buffer_index);

Functions	Results
mask8AndSet ()	/** * @fn mask8AndSet() * @brief returns the byte_value after setting the set_value into the input byte as according to * bits mentioned in the mask * @param mask -> String (Binary or Hex Mask string. Mask should always have 8 bits/1byte) * @param byte_value -> Uint8 (Input Byte Value) * @param set_value -> Integer (value to be set) * @return 8 bit integer -> The byte_value after setting the set value into it. */ BinaryUtils.mask8AndSet(mask, bytes_value, set_value);
mask16AndSet ()	/** * @fn mask16AndSet() * @brief sets the number of bits value into the input buffer from buffer index as according to * bits mentioned in the mask * @param mask -> String (Binary or Hex Mask string. Mask should always have 16 * bits/2bytes) * @param buffer -> Uint8Array[] (Input buffer) * @param buffer_index -> Integer (Start buffer index) * @param set_value -> Integer (value to be set) * @return updated input buffer */ BinaryUtils.mask16AndSet(mask, bytes_value, set_value);
mask32AndSet ()	/** * @fn mask32AndSet() * @brief sets the number of bits value into the input buffer from buffer index as according to * bits mentioned in the mask * @param mask -> String (Binary or Hex Mask string. Mask should always have 32 * bits/4bytes) * @param buffer -> Uint8Array[] (Input buffer) * @param buffer_index -> Integer (Start buffer index) * @param set_value -> Integer (value to be set) * @return updated input buffer */ BinaryUtils.mask32AndSet(mask, bytes_value, set_value);

Functions	Results
crc16 ()	/** * @fn crc16() * @brief calculates the crc16 checksum on the input byte array * @param algorithm -> Builtin Global Object [CRC16Algorithm.ARC, CRC16Algorithm.XMODEM, CRC16Algorithm.MODBUS, CRC16Algorithm.USB, CRC16Algorithm.CMS] * @param buffer -> Uint8Array (Input buffer) * @param start_index -> Integer [OPTIONAL] (Start index of input buffer) * @param length -> Integer [OPTIONAL] (length of input buffer from start index) * @return 16 bit integer -> crc16 checksum */ BinaryUtils.crc16(algorithm, buffer, start_index, length)
crc32 ()	/** * @fn crc32() * @brief calculates the crc32 checksum on the input byte array * @param algorithm -> Builtin Global Object [CRC32Algorithm.AIXM] * @param buffer -> Uint8Array (Input buffer) * @param start_index -> Integer [OPTIONAL] (Start index of input buffer) * @param length -> Integer [OPTIONAL] (length of input buffer from start index) * @return 32 bit integer -> crc32 checksum */ BinaryUtils.crc32(algorithm, buffer, start_index, length)
base64_encode ()	* @fn base64_encode() * @brief base64 encode the input byte array * @param buffer -> Uint8Array (Input buffer) * @return String -> base64 encoded string */ BinaryUtils.base64_encode(buffer); /** * @fn base64_decode() * @brief decode the base64 encoded string * @param encoded_data -> String (Input string) * @return Uint8Array -> decoded array buffer */ BinaryUtils.base64_decode(encoded_data);

Few examples of on_update functions

- Applying Bit Manipulation on Uint8Array using BinaryUtils object

```
// Buffer Input (Uint8Array) -> 0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48
0x49 0x4A
function on_update() {
// encoded_message -> QUJDREVGR0hJSg==
```

```
output.encoded_message = BinaryUtils.base64_encode(buffer);

// decoded_message -> "Cisco - Edge Intelligence"
var decoded_buf =
BinaryUtils.base64_decode("Q2lzY28gLSBFZGdlIEludGVsbGlnZW5jZQ==");
output.decoded_message = String.fromCharCode.apply(null, new
Uint8Array(decoded_buf));

// crc16_checksum -> 3230
output.crc16_checksum = BinaryUtils.crc16(CRC16Algorithm.ARC, buffer);

// crc32_checksum -> 840854789
output.crc32_checksum = BinaryUtils.crc32(CRC32Algorithm.AIXM, buffer);

output.publish();
}
```

- Temperature conversion from Celsius to Fahrenheit

```
function on_update() {
var temp_celsius = input.temperature;
var temp_fahrenheit = temp_celsius * (9 / 5) + 32;
output.temperature = temp_fahrenheit;
output.publish();
}
```

- This code snippet is responsible for processing incoming sensor data (like temperature or vibration), calculating a sliding average for that data, and then publishing the result.

```
function FastSlidingAvg(window_size) {
this.size = window_size;
this.values = [];
this.sum = 0;
this.update = function (value) {
if (this.values.length == this.size) {
this.sum -= this.values[0];
this.values.shift();
}
this.sum += value;
this.values.push(value);
};
this.avg = function () {
return this.sum / this.values.length;
};
}
var acc = {};
function init() {
acc["temp"] = new FastSlidingAvg(parameters.window_size);
acc["vibration"] = new FastSlidingAvg(parameters.window_size);
output.temp = 0;
output.vibration = 0;
}
function on_update() {
var field_name = trigger.field_name;
var value = input[field_name];
acc[field_name].update(value);
output[field_name] = acc[field_name].avg();
output.publish();
}
```

Logging

These scripts represent common logging methods which helps to emit the log messages. Using a global object named `logger`, you can emit log messages with these six different severity level APIs. They are used to record events, status, and errors within an application, with each method corresponding to a different log level or severity.

```
logger.trace(...)
logger.debug(...)
logger.info(...)
logger.warn(...)
logger.error(...)
logger.fatal(...)
```

Example:

```
logger.info("Calling script with:", my_value);
```

- This is a log entry which indicates the exact date and time when the log event occurred. Each call accepts a variable number of arguments and will print them to a log message with the following pattern:

  ```
  2014-10-17T19:26:42.141Z INF: some text 123
  ^^^^^^^^^^^^^^^^^^^^^^^^ ^^^ ^^^^^^^^^ ^^^
  \Timestamp | \arg1 \arg2
  \LogLevel
  ```

- Custom types must implement a `toString()` method to be displayed in log messages. For example:

  ```
  function MyType() {}
  MyType.prototype.toString = function () {
  return "This is MyType";
  };
  var my_type = new MyType();
  logger.info("MyType is:", my_type);
  ```

Applying Bit Manipulation `get` bit functions on Uint8Array using BinaryUtils object

Get bit functions are for reading/inspecting data (checking if a bit is `0` or `1`).

1. string_to_byte_array(str)

   - ```
     function string_to_byte_array(str) {
     var strLen=str.length;
     var buf = new Uint8Array(strLen);
     for (var i=0; i < strLen; i++) {
     buf[i] = str.charCodeAt(i);
     }
     return buf;
     }
     ```

2. on_update()

   - Main function that processes input string data and applies various bit manipulation functions from the `BinaryUtils` object.
   - Steps inside `on_update()`:
     - Convert input string to byte array using `string_to_byte_array`
     - Apply six different bit mask functions on the byte array:

       ```
       // Buffer Input (Uint8Array) -> //# 0x 4E 62 63 64 65 66
       function on_update() {
       var get_buf = string_to_byte_array(input.getbit_array_data);
       ```
       ```

Code Fucntions	Descriptions
mask8: Applies an 8-bit mask to the first byte	`//function 1. mask8` `output.getbit_mask8 =` `BinaryUtils.mask8("0b1111_0000",` `get_buf[0]);`
mask8AndShiftRight: Applies an 8-bit mask and shifts right on the first byte	`//function 2. mask8AndShiftRight` `output.getbit_mask8AndShiftRight =` `BinaryUtils.mask8AndShiftRight(` `"0b11110000",` `get_buf[0]` `);`
mask16: Applies a 16-bit mask starting at offset 0	`//function 3. mask16` `output.getbit_mask16 =` `BinaryUtils.mask16(` `"0b0000_1111_1111_0000",` `get_buf,` `0` `);`
mask16AndShiftRight: Applies a 16-bit mask and shifts right starting at offset 0	`//function 4. mask16AndShiftRight` `output.getbit_mask16AndShiftRight =` `BinaryUtils.mask16AndShiftRight(` `"0b0000_1111_1111_0000",` `get_buf,` `0` `);`
mask32: Applies a 32-bit mask starting at offset 0	`//function 5. mask32` `output.getbit_mask32 =` `BinaryUtils.mask32(` `"0b0000_1111_1111_1111_1111_1111_1111_0000",` `get_buf,` `0` `);`
mask32AndShiftRight: Applies a 32-bit mask and shifts right starting at offset 0	`//function 6. mask32AndShiftRight` `output.getbit_mask32AndShiftRight =` `BinaryUtils.mask32AndShiftRight(` `"0b0000_1111_1111_1111_1111_1111_1111_0000",` `get_buf,` `0` `);`

- Publish the output after all manipulations

```
output.publish();
```

Applying Bit Manipulation set bit functions on Uint8Array using BinaryUtils object

Set bit functions are for writing/modifying data (changing a bit to `1`).

1. string_to_byte_array(str)

- ```
  function string_to_byte_array(str) {
  var strLen=str.length;
  var buf = new Uint8Array(strLen);
  for (var i=0; i < strLen; i++) {
  buf[i] = str.charCodeAt(i);
  }
  ```
  ```

```
return buf;
}
```

2. on_update()
   - Main function that processes input string data and applies various bit manipulation functions from the `BinaryUtils` object.
   - Steps inside `on_update()`:
     - Convert input string to byte array using `string_to_byte_array`
     - Apply six different bit mask functions on the byte array:

```
// Buffer Input (Uint8Array) -> //# 0x 41 42 43 44 45 46 - "ABCDEF"
function on_update() {
var set_buf = string_to_byte_array(input.setbit_array_data);
var dup_set_buf = set_buf;
```

Code Fucntions	Descriptions
mask8AndSet	`//function 1. mask8AndSet ,output: 0x71 = 113` `output.setbit_mask8AndSet = BinaryUtils.mask8AndSet("0xF0", set_buf[0], 7);`
mask16AndSet	`//function 2. mask16AndSet ,output: "aBCDEF"` `set_buf = dup_set_buf; BinaryUtils.mask16AndSet("0xFF00", set_buf, 0, 0x61);` `output.setbit_mask16AndSet = String.fromCharCode.apply( null, new Uint8Array(set_buf) );`
mask32AndSet	`//function 3. mask32AndSet ,output: "bCDDEF"` `set_buf = dup_set_buf; BinaryUtils.mask32AndSet("0x0F_FF_FF_F0", set_buf, 0, 0x243444);` `output.setbit_mask32AndSet = String.fromCharCode.apply( null, new Uint8Array(set_buf) );`

     - Publish the output after all manipulations

```
output.publish();
```

Currently, all levels are treated equally and either are all emitted or none.

- A pipeline set to `productive = true` will never emit any log messages.
- A pipelines set to `productive = false` will emit all messages.

All emitted messages are subject to quota handling.

Function `mqtt_add_subscription` api from data logic for MQTT destination

```
/**
* @fn mqtt_add_subscription()
* @brief it is to add the topic subscription for mqtt link the API should
be called from data
* logic init() function and its applicable only for mqtt link, the
API is a blocking * call and will block current thread for maximum
100ms
* @param topic: valid mqtt topic
* @param qos: its optional param allowed values (0,1 or 2) default value
is considered 0 if not * provided
* @return none on success
* @Error message for the invalid use
* 1. Wrong number of argument:
* Error message: Error executing JavaScript: TypeError:
mqtt_add_subscription(): wrong number of argument provide, expecting 1
(topic) or 2 arguments (topic,qos)
* 2. non-utf in topic:
* Error message: Error executing JavaScript: TypeError:
mqtt_add_subscription(): Topic is not UTF-8 string
* 3. invalid QOS
* Error message: Error executing JavaScript: RangeError:
mqtt_add_subscription(): QoS should be 0,1 or 2
* 4. commandPath topic not allowed
* Error message: Error executing JavaScript: SyntaxError:
mqtt_add_subscription(): topic 'commandPath' cannot be * subscribed from
data logic script
* 5. not supported destination
* Error message: Error executing JavaScript: TypeError:
mqtt_add_subscription(): api is only supported for mqtt destination
* 6. api mqtt_add_subscription() is allowed only in init()
* Error message: Error executing JavaScript: TypeError:
mqtt_add_subscription(): api is only allowed to be called from init()
*
*/
mqtt_add_subscription(topic,qos)
```

HTTP utilities

The global object http provides HTTP client functionality for making web requests from JavaScript. All HTTP methods are asynchronous and use error-first callbacks.

HTTP request methods

```
/**
* @fn http.get(url, [headers], callback)
* @brief Performs an HTTP GET request
*/
function handleGetResponse(err, res) {
if (err) {
logger.error("Request failed: " + err.message);
} else {
output.api_data = JSON.parse(res.body);
output.publish();
}
}

http.get("https://api.example.com/data", {"Accept": "application/json"},
handleGetResponse);

/**
* @fn http.post(url, body, [headers], callback)
```

```
* @brief Performs an HTTP POST request
*/
function handlePostResponse(err, res) {
if (err) {
logger.error("POST failed: " + err.message);
} else {
logger.info("Created: " + res.statusCode);
}
}

var data = {"name": "John", "age": 30};
http.post("https://api.example.com/users", data, {"Content-Type":
"application/json"}, handlePostResponse);

/**
* @fn http.put(url, body, [headers], callback)
* @brief Performs an HTTP PUT request (same parameters as POST)
*/

/**
* @fn http.delete(url, [headers], callback)
* @brief Performs an HTTP DELETE request (same parameters as GET)
*/
```

HTTP configuration

```
/**
* @fn http.setTimeout(milliseconds)
* @brief Sets the default timeout for HTTP requests
*/
http.setTimeout(30000); // 30 seconds

/**
* @fn http.setConnectionReuse(enabled)
* @brief Controls whether HTTP connections are reused or closed after each
request
*/
http.setConnectionReuse(false); // Disable for problematic servers

/**
* @fn http.getRequestCount()
* @brief Returns the current number of pending HTTP requests
*/
var pendingRequests = http.getRequestCount();
```

SSL configuration

```
/**
* @fn http.setSSLOptions(options)
* @brief Configures SSL/TLS options for HTTPS requests
*/
http.setSSLOptions({
verify: true, // Enable peer verification
verifyHostname: true, // Enable hostname verification
allowSelfSigned: false, // Allow self-signed certificates
caFile: "/path/to/ca.pem" // CA certificate file path or content
});

/**
* @fn http.setVerifyMode(verify)
* @brief Sets the SSL peer verification mode
*/
http.setVerifyMode(false); // Disable SSL verification for testing
```

```
/**
* @fn http.allowSelfSignedCertificates(allow)
* @brief Controls whether self-signed certificates are accepted
*/
http.allowSelfSignedCertificates(true); // Allow for development
```

Response object

The response object passed to callbacks contains:

- `statusCode` (Number): HTTP status code
- `body` (String): Response body
- `headers` (Object): Response headers

Error handling

All HTTP methods use error-first callbacks. The error object contains:

- `message` (String): Error description
- `code` (String): Error code (if available)

HTTP usage example

```
function handleApiResponse(err, res) {
if (err) {
logger.error("HTTP error: " + err.message);
return;
}

if (res.statusCode === 200) {
var data = JSON.parse(res.body);
output.external_data = data;
output.publish();
} else {
logger.warn("HTTP status: " + res.statusCode);
}
}

function init() {
http.setTimeout(30000);
http.setConnectionReuse(true);
http.setVerifyMode(true);
}

function on_update() {
var apiUrl = "https://api.example.com/sensor/" + input.sensor_id;
http.get(apiUrl, {"Accept": "application/json"}, handleApiResponse);
}
```

Protobuf utilities

The `protobuf` module provides Protocol Buffer encoding and decoding functionality. Protocol Buffers (protobuf) is a language-neutral, platform-neutral extensible mechanism for serializing structured data. The module is accessed via `require("protobuf")`.

Protobuf best practices

1. **Initialize in init()**: Always initialize protobuf types in the `init()` function to avoid repeated parsing
2. **Memory Management**: Store protobuf types in global objects to reuse across function calls
3. **Base64 Encoding**: Proto files must be base64 encoded when embedded in JavaScript

Example: simple message encoding/decoding

```
var pb = new Object();
function init() {
var protobuf = require("protobuf");
// Base64 encoded proto file content for a simple message
// Original proto:
// syntax = "proto3";
// package simple;
// message SimpleMessage {
// string name = 1;
// int32 value = 2;
// }
var protofile =
"c3ludGF4ID0gInByb3RvMyI7CnBhY2thZ2Ugc2ltcGxlOwptZXNzYWdlIFNpbXBsZU1lc3NhZ2
UgewogIHN0cmluZyBuYW1lID0gMTsKICBpbnQzMiB2YWx1ZSA9IDI7Cn0=";
var root = protobuf.init([protofile]);
pb.SimpleMessage = root.lookupType("simple.SimpleMessage");
}

function on_update() {
// Create and encode a protobuf message
var msg = pb.SimpleMessage.create({
name: input.device_name,
value: input.sensor_reading
});
var encoded_buffer = pb.SimpleMessage.encode(msg).finish();
publish("output", encoded_buffer);

// Decode a protobuf message
var decoded_msg = pb.SimpleMessage.decode(encoded_buffer);
output.decoded_name = decoded_msg.name;
output.decoded_value = decoded_msg.value;
}
```

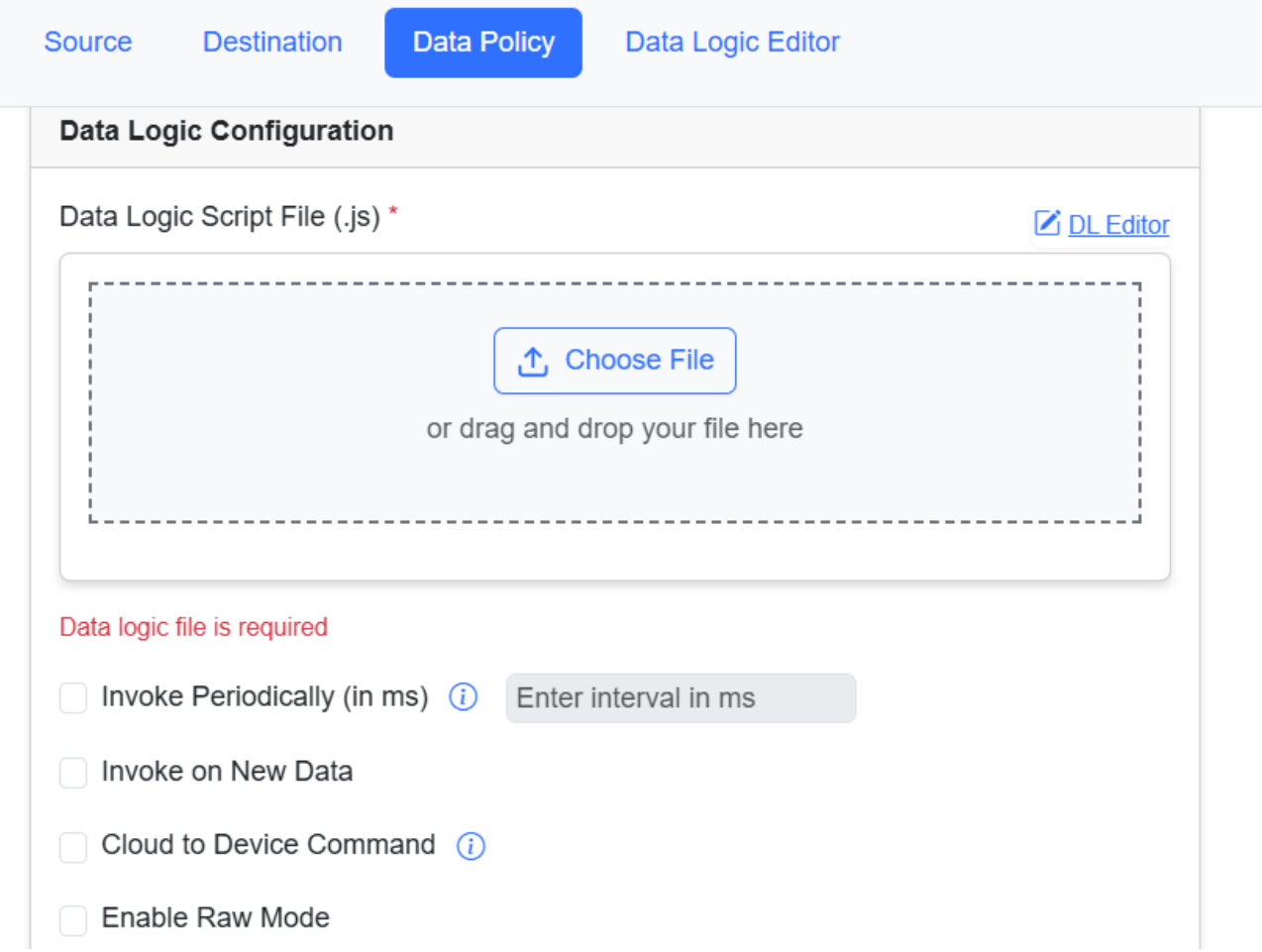
Create a Data Logic

Data logic is used to transform data from connected assets before it is delivered to a destination. Unlike Data Rules that send all the raw data for an Asset Type, Data Logic allows you to aggregate or average data, send only data that exceeds certain value, detect anomalies, and more.

Data Logic is developed and debugged using JavaScript in Microsoft Visual Studio (VS Code) and embedded UI editor. Scripts are synchronized to Cisco IoT where they can be deployed to EI Agents running on Edge Devices.

Procedure

Step 1 From the **Data Policy** tab, select **Data Logic** checkbox.

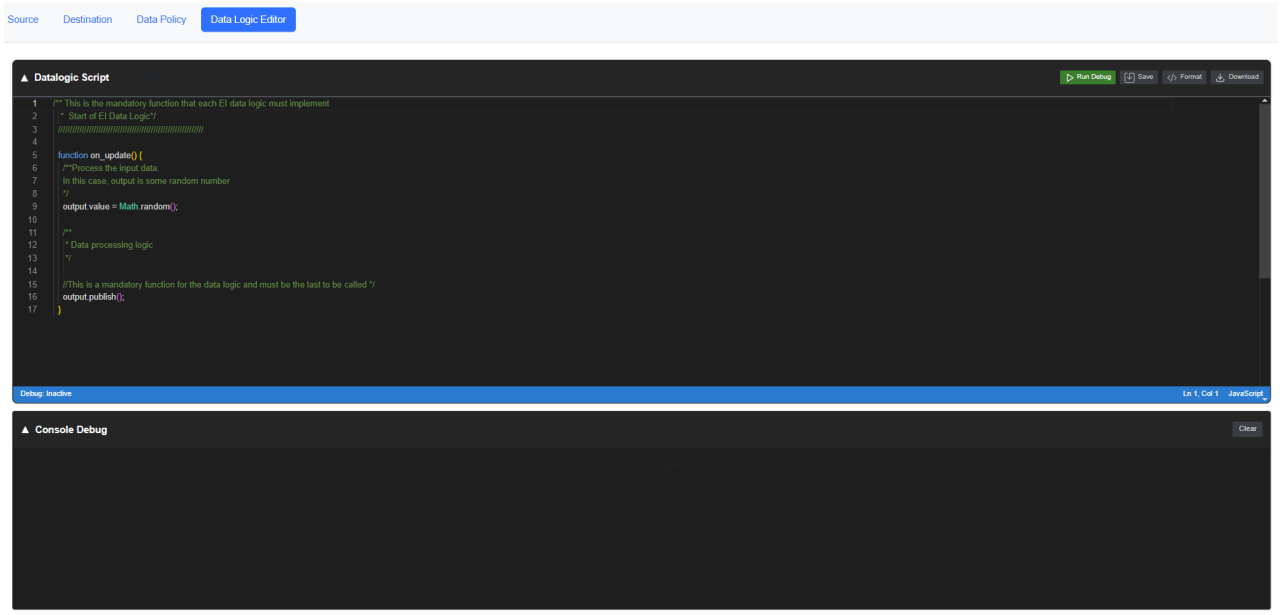


Note
Clicking upon **Data Logic** radio button the **Data Logic Editor** tab appears next to Data Policy.

Step 2 To add a data logic script, click **Choose File** to select precreated new script (example: **LogicExample.js**) from your local drive.

Step 3 To review or to create a new data logic script, click **DL Editor**.

- Data logic functionalities where user can write his own transformation code to help to transform data from source to destination.
- It helps to verify and validate the existing script file by clicking **Run Debug**.
- You can write a new script and validate it.



Note
HTTP funtionality is introduced on data logic script. Data logic editor supports HTTP CRUD (create, retrive, update and delete) opeartion. To know various supported HTTP features, refer to HTTP and HTTPS protocols in data logic scripts, on page 53.

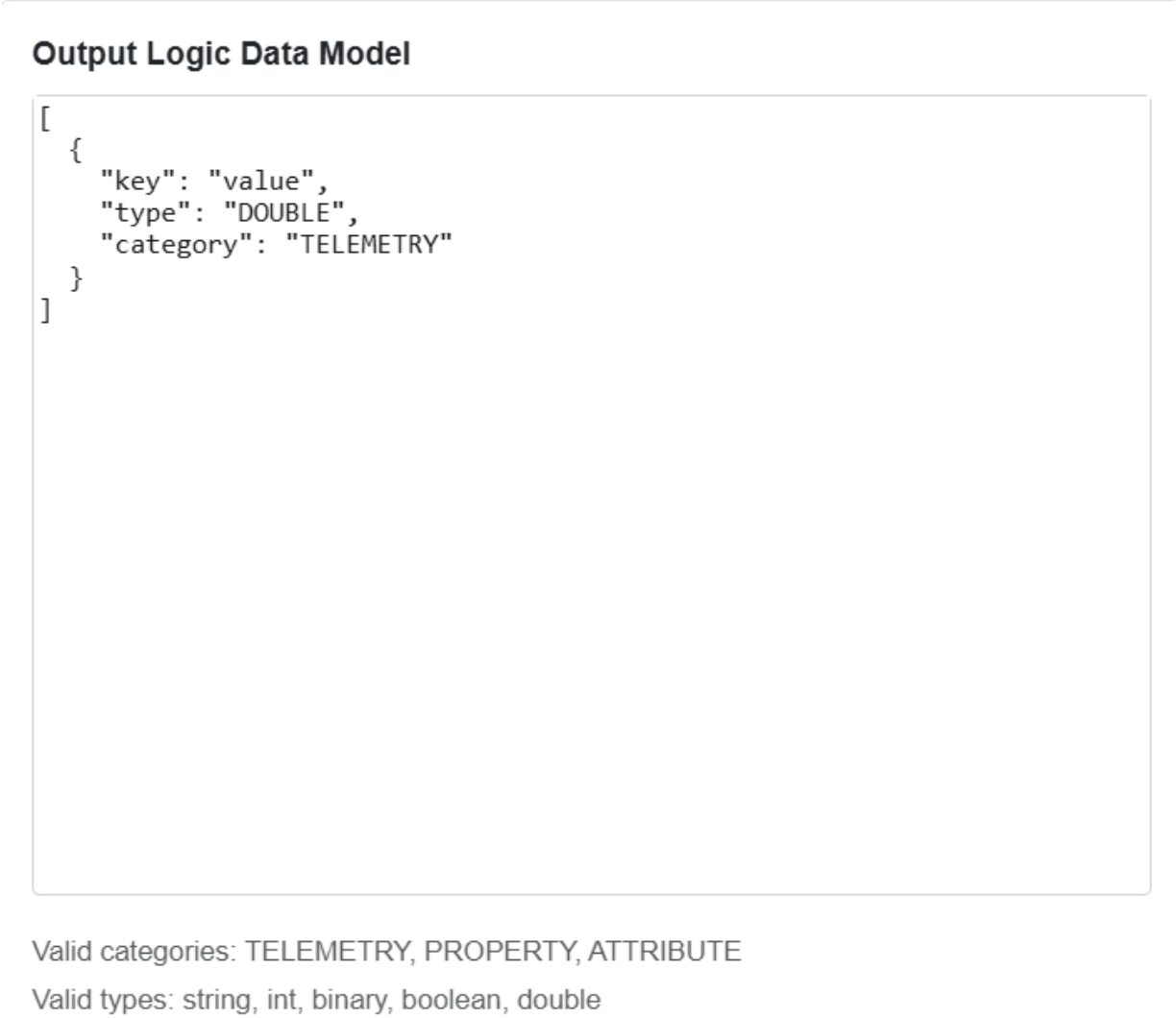
Step 4 After you upload a script file, the new download and delete icon appears. It helps to check/validate what uploaded from other agents. Another user can also check an existing pipeline and can download it.

Step 5 From the following run-time options, select when the script will be run.

- **Invoke Periodically (in ms)**: when you tick the checkbox, it enables the ms interval. Enter the interval time, in ms. For example, if you enter 500, the Data Logic script will be called every 500 ms. Enable this option if needed.
- **Invoke on New Data**: The script is called when data changes.
- **Cloud to Device Command**: This function is called when you receive a command from the cloud.
- **Enable Raw Mode**: A default output JSON data model is automatically created when a Data Logic script is created.
- (Optional) In the **Output Logic Data Model**, you can modify the default format of the Output Data Logic Model script (in JSON) and specify the output model with custom names.

  Valid categories: TELEMETRY, PROPERTY, ATTRIBUTE

  Valid types: string, int, binary, boolean, double



HTTP and HTTPS protocols in data logic scripts

The data logic editor supports HTTP and HTTPS protocols. For these protocols, you can use the CRUD operations, configure SSL settings, timeout settings, and cancel requests.

Cisco devices use the following ports by default:

protocol	Device port
HTTP	80
HTTPS	443

If a device uses the default ports for HTTP and HTTPS communications, the configurations in the data logic script are automatically applied to the port. If a device uses a different port for these protocols, additional configurations may be required.

Secure the HTTP and HTTPS communications by using one of these TLS configurations.

- Trusted Certificate Authority (CA)
- Self-signed certificates
- Self-signed CA
- Mutual TLS (mTLS) using client certificate and key
- Use the API (`setSslOptions`) to configure the root certificate, client certificate, and client key

Method	Signature	Sample Snippet
GET	/** HTTP GET: Fetch all items Signature: http.get(url: string, headers: object, callback: function) Example: Get list of all items Expected: 200 OK with array of items */	function getAllItemsCallback(err, status, body, headers) { if (err) { logger.error("[GET ALL] Error:", err.message); } else { logger.info("[GET ALL] Success:", status, body); } } http.get("http://localhost:3000/a pi/data", { "Accept": "application/json" }, getAllItemsCallback);
POST	/*** HTTP POST: Create new item* Signature: http.post(url: string, body: object|string, headers: object, callback: function)* Example: Create new item with name "Item Three" * Expected: 201 Created */	function createItemCallback(err, status, body, headers) { if (err) { logger.error("[POST] Error:", err.message); } else { logger.info("[POST] Success:", status, body); } } http.post("http://localhost:3000/a pi/data", { name: "Item Three" }, { "Content-Type": "application/json" }, createItemCallback);

Method	Signature	Sample Snippet
DELETE	/** * HTTP DELETE: Delete item by ID * Signature: http.delete(url: string, callback: function) * Example: Delete item with ID 2 * Expected: 200 OK or 404 Not Found */	function deleteItemCallback(err, status, body, headers) { if (err) { logger.error(" [DELETE] Error:", err.message); } else { logger.info("[DELETE] Success:", status, body); } } http.delete("http://localhost:3000/api/data/2", deleteItemCallback);
PUT	/** * HTTP PUT: Update item by ID * Signature: http.put(url: string, body: object|string, headers: object, callback: function) * Example: Update item ID 1 to have a new name * Expected: 200 OK or 404 Not Found */	function updateItemCallback(err, status, body, headers) { if (err) { logger.error("[PUT] Error:", err.message); } else { logger.info("[PUT] Success:", status, body); } } http.put("http://localhost:3000/api/data/1", { name: "Updated Item One" }, { "Content-Type": "application/json" }, updateItemCallback);
SSL Options	/** * SSL Configuration Example (optional) * Signature: http.setSSLOptions(options : object) * Example: Enable selfsigned certificate support * Use this before calling HTTPS endpoints */	http.setSSLOptions({ verify: true, verifyHostname: true, // allowSelfSigned: true, // });

Method	Signature	Sample Snippet
SSL options with Certificates	Certificates and keys can also be provided via the `customAttribute` file options. These values can be integrated into the data logic. Before being passed to `setSslOptions`, the certificate files must be base64- encoded, as demonstrated in the provided example.	`function decodeFileContent(fileAttribute) { // File attributes come as binary data, need to decode to string var decoder = new TextDecoder(); return decoder.decode(fileAttribute); }` `var caCert = decodeFileContent(input.ca_certificate); var clientCert = decodeFileContent(input.client_certificate); var clientKey = decodeFileContent(input.client_key); var sslOptions = { verify: true, caFile: caCert, // root certificate verifyHostname: false, certFile: clientCert, // client certificate keyFile: clientKey // client / private key };`  `var sslConfigured = http.setSSLOptions(sslOptions);`
Timeout	`/**` `Timeout Configuration` `Signature: http.setTimeout(timeoutMs: number)` `Example: Set timeout to 2000 ms */`	`http.setTimeout(2000); // Set timeout for all requests to 2 seconds MIN_TIMEOUT =1000; // 1 second MAX_TIMEOUT =300000; // 5 minutes DEFAULT_TIMEOUT = 8000; // 8 seconds`

Method	Signature	Sample Snippet
Connection Reuse	/* http.setConnectionReuse(enabled : boolean): boolean Controls whether HTTP connections are reused (pooled) or closed after each request. □ Parameters enabled (boolean): true: Enables connection reuse (default behavior) false: Disables connection reuse — HTTP connection will be closed after each request □□ Returns true on success false if setting failed (e.g., unsupported in the current environment)*/	function init() { http.setConnectionReuse(true); // by dfault its true mentioned this . }

Known Limitations

- Avoid using `http.get/post/delete/put` within the `init()` call to prevent unexpected delays during initialization.
- When making API calls that handle large payloads, particularly for CRUD operations, refrain from logging the full response due to the logging buffer's size limit.
- Configure SSL options during the `init` call to ensure secure communication is established from the start.
- Be aware that the maximum payload size is limited to 1MB.

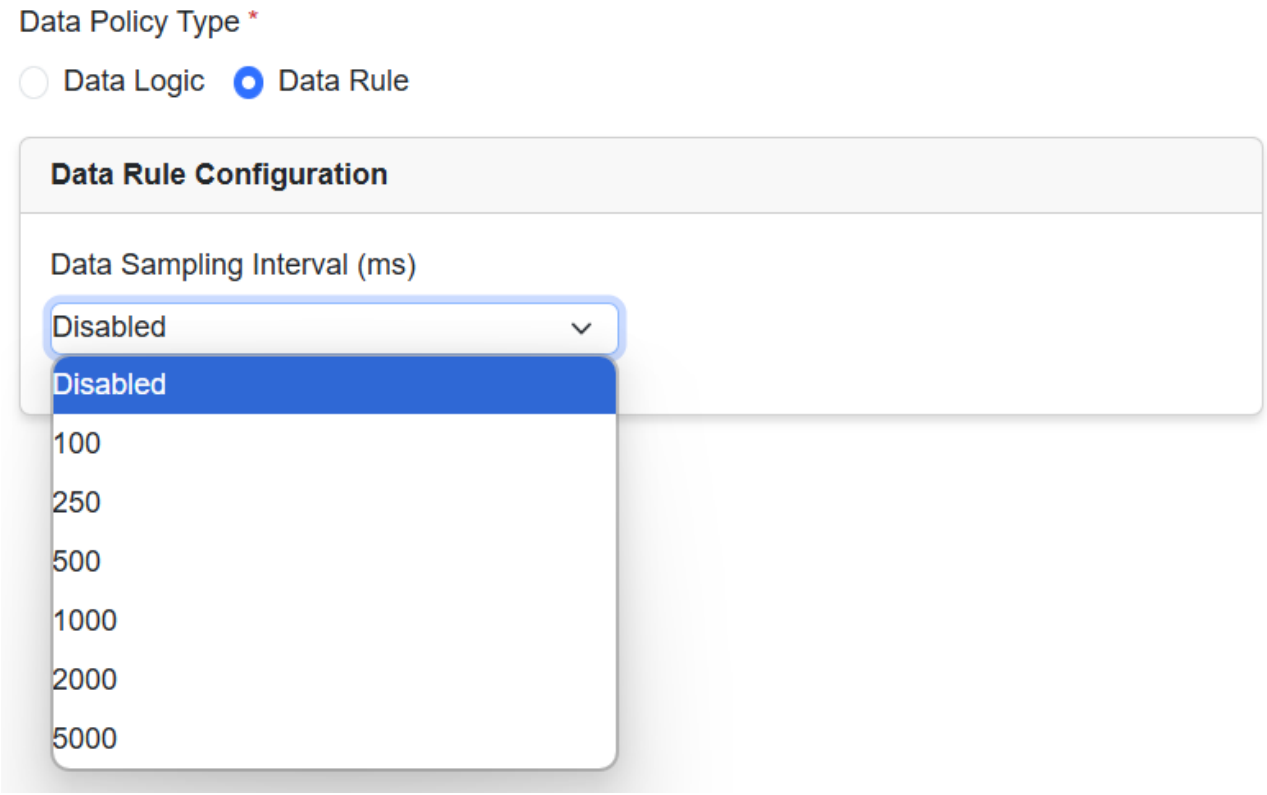
Create a Data Rule

Data rules define the flow of data, from connected assets to data destinations, without transformation.

Procedure

Step 1 From the **Data Policy** tab, select **Data Rule** checkbox.

Step 2 To configure the Data Rule, select a data from **Data Sampling Inerval (ms)** drop-down list.



Deploy or undeploy pipelines

When deployed, a pipeline runs on the Cisco network device where the EI agent is installed.

Procedure

Step 1 To deploy a pipeline:

a) Create a pipeline.

b) Click **Deploy**.

The deployed pipeline is listed in the **Pipelines** area of the Cisco Edge Intelligence page.

Step 2 To undeploy a pipeline, in the **Pipeline** section, select a pipeline and click **Undeploy**.

When you undeploy a pipeline, it is entirely removed from Cisco Edge Intelligence. If you wish to retain a copy of the configuration, save the pipeline configuration as a template before you undeploy the pipeline.

View health status

To track the health status of an EI agent, select a pipeline from the list of pipelines.

Procedure

Step 1 From the left pane, click any **Pipeline** for which you want to view the details.

Step 2 Click **Health Status** tab. A combined list log of pipeline overview, source status, and destination status appears for the EI agent.

Step 3 Click on any status tab to preview each log report.

Column Header	Description
Health Status	Displays the overall health of the data pipeline, indicating errors in source or destination connections.
Pipeline Status	Shows the current status of the data pipeline. For example, Error and so on.
Source Status	Indicates the connection status (online or offline) of the source asset.
Destination Status	Reflects the status (online or offline) of the data destination.